1985

Design, analysis and implementation of the primary operation, retrieve-common, of the multi-backend-database system (MBDS).

Tung, Hsiang-Lung.

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

DESIGN, ANALYSIS AND IMPLEMENTATION
OF THE PRIMARY OPERATION, RETRIEVE-COMMON,
OF THE MULTI-BACKEND DATABASE SYSTEM (MBDS)

by

Hsiang-Lung Tung

June 1985

Thesis Advisor: David K. Hsiao

Approved for public release; distribution is unlimited

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER	2. GOVT ACCESSION NO. / 3. RECIPIENT'S CATALOG NUMBER
4. TITLE *(and Subtitle)*: Design, Analysis and Implementation of the Primary Operation, Retrieve-Common, of the Multi-Backend Database System (MBDS)	5. TYPE OF REPORT & PERIOD COVERED: Master's Thesis, June 1985 / 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s): Hsiang-Lung Tung	8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Naval Postgraduate School, Monterey, CA 93943-5100	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS: Naval Postgraduate School, Monterey, CA 93943-5100	12. REPORT DATE: June 1985 / 13. NUMBER OF PAGES: 148
14. MONITORING AGENCY NAME & ADDRESS *(if different from Controlling Office)*	15. SECURITY CLASS. *(of this report)*: UNCLASSIFIED / 15a. DECLASSIFICATION/ DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT *(of this Report)*

Approved for public release; distribution is unlimited

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

18. SUPPLEMENTARY NOTES

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

Database System

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

The multi-backend database system (MBDS) in the Laboratory for Database System Research at the Naval Postgraduate School is designed to overcome the performance-gain and capacity-growth problems of either the traditional database system or the single-bakend-software database system. The original MBDS supported four primary operations, namely, RETRIEVE, DELETE, UPDATE and INSERT.
(Continued)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

S N 0102-LF-014-6601

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

ABSTRACT (Continued)

This thesis presents the design and implementation of the fifth primary operation, the RETRIEVE-COMMON operation. The retrieve-common operation is used to merge two files by their common attribute values. First, the overall design and implementation of MBDS is reviewed. Then, several alternatives are compared and analyzed to select the best one as our design and implementation approach. Finally, we describe the detailed design and the implmentation. Our goal is to maximize the utilization and minimize the effects to the existing system.
For integrating our design into MBDS, several modifications are made. The algorithms for the modifications and their program specifications are also provided in Chapter IV, V and Appendices.

Design, Analysis and Implementation of the Primary Operation, Retrieve-Common, of the Multi-Backend Database System (MBDS)

by

Hsiang-Lung Tung
Commander, Republic Of China Navy
B.S., Chinese Naval Academy, 1974

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June 1985

ABSTRACT

The multi-backend database system (MBDS) in the Laboratory for Database System Research at the Naval Postgraduate School is designed to overcome the performance-gain and capacity-growth problems of either the traditional database system or the single-backend-software database system. The original MBDS supported four primary operations, namely, RETRIEVE, DELETE, UPDATE and INSERT.

This thesis presents the design and implementation of the fifth primary operation, the RETRIEVE-COMMON operation. The retrieve-common operation is used to merge two files by their common attribute values. First, the overall design and implementation of MBDS is reviewed. Then, several alternatives are compared and analyzed to select the best one as our design and implementation approach. Finally, we describe the detailed design and the implementation. Our goal is to maximize the utilization and minimize the effects to the existing system.

For integrating our design into MBDS, several modifications are made. The algorithms for the modifications and their program specifications are also provided in Chapter IV, V and Appendices.

TABLE OF CONTENTS

LIST OF FIGURES

I. INTRODUCTION

A. THE SCOPE OF THE THESIS

A database, is a collection of stored operational data; and a database system is a computer-based system whose overall purpose is to record and maintain information (data) [Ref. 1]. The traditional approach to manage the database system is to run the database system software as an application program in a mainframe computer system. The database system must share the use and the control of the mainframe computer resources with all of the other applications of the computer system. The performance of this approach suffers whenever there is an increase from either the usage of the computer system or the database applications.

One solution to this problem is to offload the database system from the mainframe to a single, dedicated backend computer. The backend computer has its own disk storage and used to perform database operations exclusively. [Refs. 2,3]. This approach is known as the single software backend approach. Database systems based on this approach are referred to as software single backend database systems. However, this approach still has the disadvantage, that is, performance upgrades will require the replacement of the backend and this may entail software modifications and hardware disruption [Ref. 4 : p. 4].

A second approach to solve the database performance problem is to develop a special-purpose database machine with specially designed hardware. However, the cost-effectiveness of this approach, known as the hardware backend approach, has not yet been demonstrated [Ref. 5].

In order to overcome the performance-gain and capacity-growth problems of either the traditional database system or the single backend software system, a research of a multi-backend database system, known as MBDS, is conducted in the Laboratory for Database Systems Research, at the Naval Postgraduate School. Instead of a single backend computer, MBDS uses several identical (both in hardware and in software) minicomputers as its backend computers in a parallel fashion in order to gain performance gain and capacity growth. These backends with their respective disk systems are connected with another minicomputer, called the backend controller. The controller is responsible for supervising the execution of parallel database operations on the backends and for interfacing with the hosts and the user. Users access the system either by way of the host or through the controller directly (as shown in Figure 1.1).

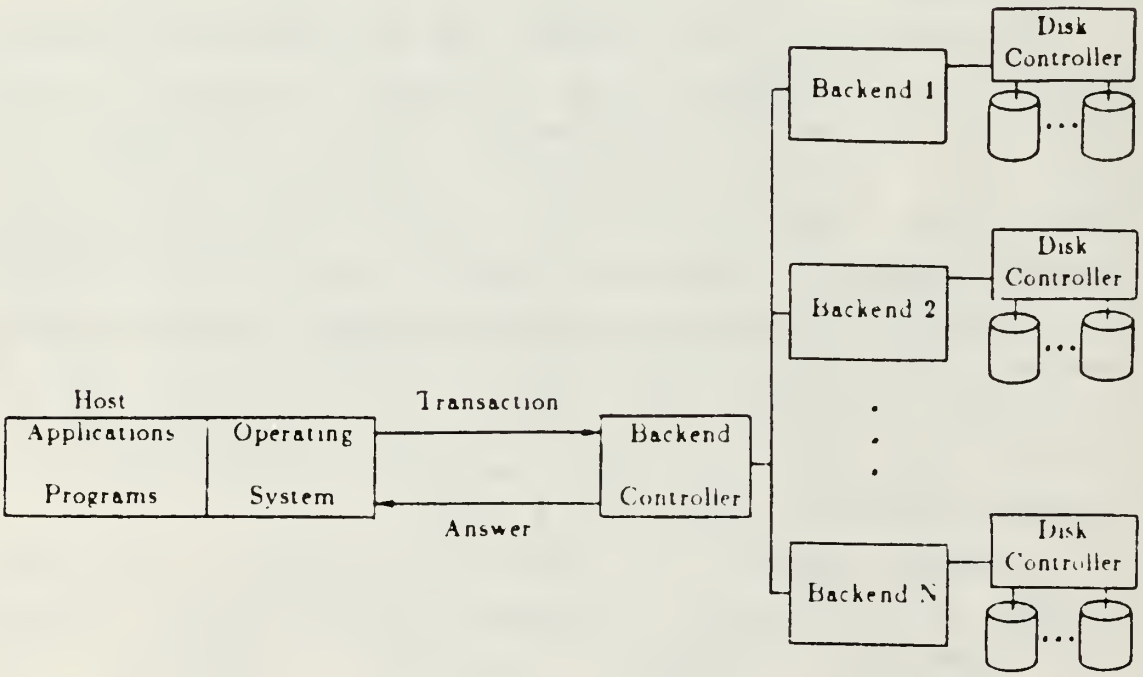


Figure 1.1 The Multi-Backend Database System (MBDS).

The attribute-based data language (ABDL) [Ref. 6] is used as the basis of the data language of MBDS. Currently, ABDL supports four primary database operations, RETRIEVE, DELETE, UPDATE and INSERT. The functions of these four database operations are shown in Figure 1.2.

Operation	Function
RETRIEVE	Retrieve records from the database
DELETE	Delete records from the database
UPDATE	Modify records of the database
INSERT	Insert records into the database

Figure 1.2 The Functions of the Current MBDS Database Operations.

In order to make MBDS a more complete database system, the fifth operation, the RETRIEVE-COMMON operation which is used to merge two files by common attribute values, has been proposed [Ref. 7]. This thesis will focus on the design and implementation of the RETRIEVE-COMMON operation of MBDS. We will propose several alternatives of the design and implementation strategies, then evaluate and analyze these alternatives based on the time complexities, the affects to the existing system and the design-goals of MBDS. According the results of the analysis, we will choose the best alternative to design and implement the fifth operation.

B. THE ORGANIZATION OF THE THESIS

The rest of this thesis is organized as follows. In chapter II we give an overview of the architecture of the MBDS. We will describe the design goals, the underlying and intended hardware, the process structure, the data model and the data language of MBDS. In chapter III, we first define the intended operation and the syntax of RETRIEVE_COMMON operation, and then evaluate and analyze the alternatives for the design and implementation. According to the analysis, we will select the best alternative to add the retrieve-common operation into the MBDS. In chapter IV, we present the details of the design for the selected approach. We also consider the possible effects of this approach to the existing system. In chapter V, we describe how to incorporate our design into MBDS. Our goal is to minimize the effects of the implementation. Finally, this thesis is summarized and concluded in chapter VI. It is hoped that this thesis will provide a definite help to the future work on MBDS.

II. THE MULTI-BACKEND DATABASE SYSTEM (MBDS)

In this chapter we will briefly review the configuration and the theory of operations of the MBDS. Most of the information provided in this chapter has been extracted from [Refs. 4,7 : pp. 1-68, 7-20]. The interested readers are encouraged to refer to the references.

A. THE SYSTEM GOALS

As mentioned in chapter I, MBDS is designed to overcome the performance problems and upgrade issues of the traditional mainframe-based or the software single-backend database system. In other words, the overall goal for MBDS is to prove that:

(1) the system is easily extensible; and

(2) the performance gain and improvement should be proportional to the multiplicity of processing and storage elements [Ref. 4 : pp.1-5].

In order to achieve the aforementioned goal, the design requirements and their correlated design issues for designing and implementing MBDS have been defined in [Ref. 7 : pp. 7-10].

1. Design Requirements

There are three main design requirements for MBDS.

(1) The system must be expandable.

(2) Both the hardware and software are generic.

(3) The database is evenly distributed across the disk systems of the backends, and, for operation, there are parallel and concurrent processing of transactions by the backends.

The first two design requirements can support the addition of backends for performance enhancement and capacity growth by adding new backends of the same type and by using existing system software. With the third requirement, performance gain (in terms of response-time reduction) and capacity growth (in terms of response-time invariance) of the system are likely to be in proportion to the number of backends of the system.

2. Design Issues

There are several issues which must be resolved in order to meet the design requirements of MBDS. The first issue concerns the backend controller. As shown in Figure 1.1, the controller may become a primary bottleneck of the system. In order to avoid this problem, the functions of the controller should be minimized and reduced to the pre-processing of the user transactions, the post-processing of the transaction results, the sending and receiving data between the backends and the host, and the arbitration of data insertion into the database.

The second design issue addresses the characteristics and functionality of the communication bus between the controller and the backends. The bus should be cost-effective and efficient for both backend communication and backend addition.

The third class of issues involves the backends of the system. The backends must have identical software to allow replication of the software on a new backend. Additionally, the backends must have complete software to perform all of the database management functions. These functions include directory management, concurrency control, record processing and communication.

The fourth design issue concerns the database. The database should be evenly distributed across all the disk systems of the backends.

The fifth design issue is on the choice of a data model and data language. The data model should easily support the required data distribution and the data placement of the database. The data language for the system is of course based on the chosen data model. It must capture all of the primary operations of the database system. The chosen data model is the attribute-based data model and the data language is the attribute-based data language.

The sixth design issue focuses on minimizing the communications traffic of the system. The controller should only communicate with the backends for sending the pre-processed user transaction, for arbitrating the data placement, and for receiving results. The backends should only communicate with the controller for sending the results of the user transactions. Communication among backends should be held to a minimum.

The seventh issue deals with the directory placement strategies. In order to enable each backend to perform all the database management functions and minimize the communication among backends, the directory data are duplicated at each backend.

B. THE UNDERLYING AND INTENDED HARDWARE

An overview of MBDS hardware organization is shown in Figure 2.1 User access is accomplished through a host computer which in turn communicates with the controller. When a transaction (either a request or a set of requests) is received, the controller will broadcast the transaction to all the backends. Since the data of all data files are evenly distributed across all the backends, all backends can now execute the same request in parallel. A queue of requests is maintained in each backend. When a backend

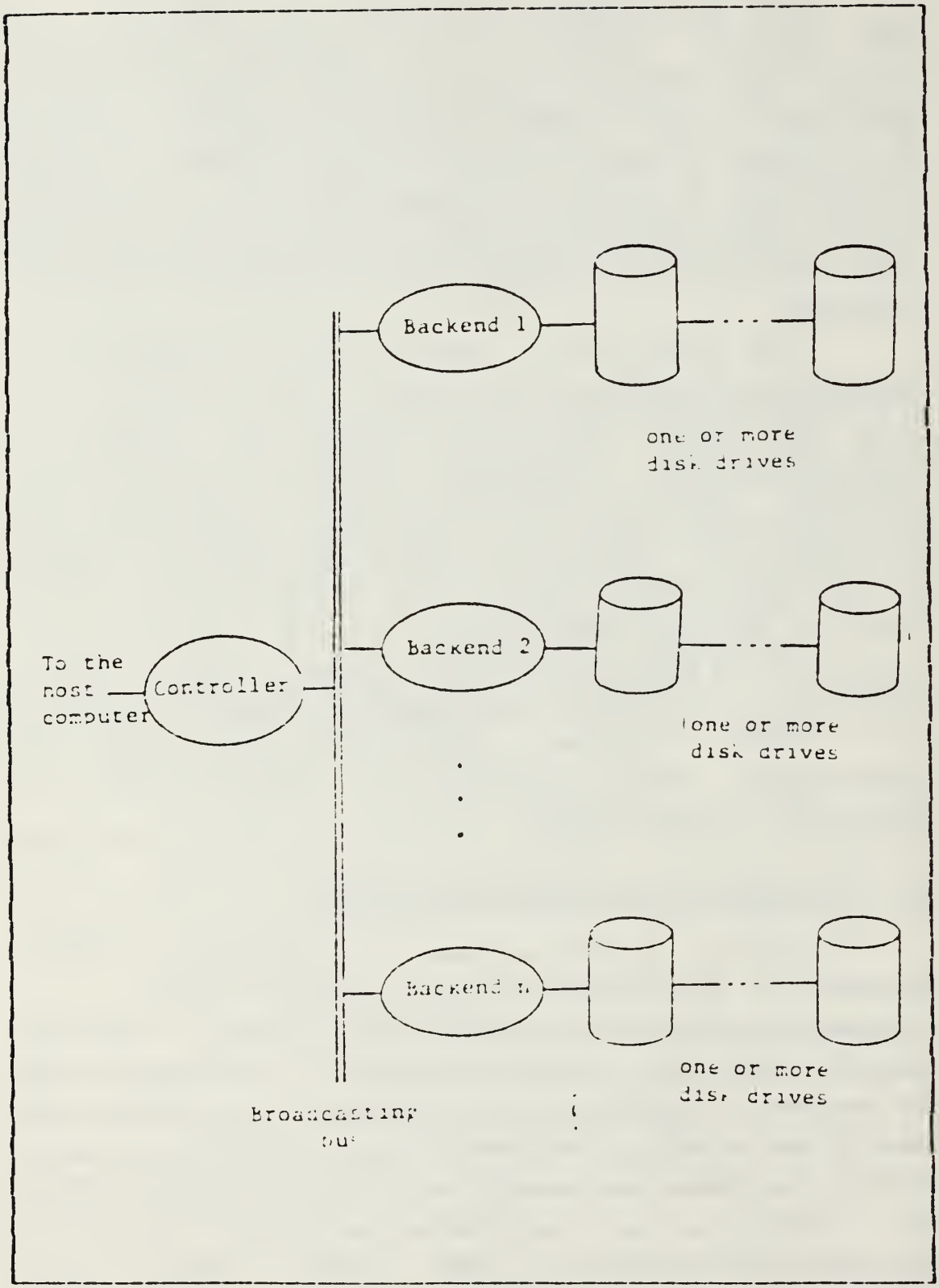


Figure 2.1 The MBDS Hardware Organization.

finishes executing one request it will send the results of that request to the controller and be able to start executing the next request independent to the other backend.

Originally, MBDS is designed to be configured with a number of microprocessor-based processing units and their disk subsystems and be connected by a broadcast-based communications line. When the implementation of MBDS began, neither the microprocessor-based computers nor the broadcast-based communications devices were available. The present MBDS is configured with a VAX-11/780 (VMS OS) as both the host and the controller and two PDP-11/44s (RSX-11M OS) and their disk systems as the backends. Communication between computers is accomplished by time-division-multiplexed buses, knowns as parallel communication links (PCLs). The broadcasting bus is simulated by the PCL.

Currently, MBDS is being down-loaded to an initial configuration of eight microprocessor-based, broadcast-bus-connected, and Winchester-drive-supported workstations, with one of the eight being used as the controller and the others as the backends. This workstation (Sun-2/170, 4.2 BSD UNIX OS) has the Motorola MC68010 as the CPU with 16 mbytes of virtual space per process and uses Ethernet as the broadcast bus among workstations. The disk drives on the backends are Fujitsu Eagle Winchester-type drives, with a formated capacity of 380 mbytes per drive.

C. THE DATA MODEL AND THE DATA LANGUAGE

In this section we will first introduce the concept and terminology of the attribute-based data model which is the data model used in MBDS, then describe the data language in which users may issue request to MBDS.

1. The Attribute-based Data Model

MBDS chooses the attribute-based data model to be its data model. In the attribute-based data model, data is modeled with the constructs: database, file, record, attribute-value pair (keyword), directory keyword, directory, record body, keyword predicate, and query. Informally, a database is a collection of files, each file contains a groups of records which are characterized by a unique set of directory keywords. A record is composed of two parts. The first part is a collection of attribute-value pairs or keywords. An attribute-value pair is a member of the Cartesian product of the attribute name and the value domain of the attribute. As an example, <SALARY, 30000> is an attribute-value pair having 30000 as the value for the attribute SALARY. All the attributes in a records are required to be distinct. Certain attribute-value pairs of a record (or a file) are called the directory keyword of that record (file), because either the attribute-value pairs or the ranges of their attribute values are kept in the directory for addressing the record (file). The rest of the record is textual information which is referred to as the record body.

The angle brackets, <, >, enclose an attribute-value pair. The curly brackets, {, }, include the record body. The parenthesis, (,), form a record. The first attribute-value of all records of a file is the same. In particular, the attribute is FILE and the value is the file name. An example of a record of employee file is shown below:

(<FILE, Employee>, <JOB, Mgr>, <DEPT,Toy>, <SALARY, 30000>
{Employee Description})

The record has four keywords and a record body of employee description.

A keyword predicate, or simply predicate, is of the form

(attribute, relational operator, value).

Without confusion, we also use parenthesis to enclose a predicate. A relational operator can be one of (=, !=, <, =<, >=). For example, (SALARY > 20000) is a predicate. A keyword K is said to satisfy a predicate T if the attribute of K is identical to the attribute in T and the relation specified by the relational operator of T holds between the value of K and the value in T. For example, the keyword <SALARY, 30000> satisfies the predicate (SALARY > 20000).

A query consists of several keyword predicates in disjunctive normal form. An example of a query is:

((DEPT=Toy) and ((SALARY<30000) or (SALARY>20000))).

2. The Attribute-based Data Language

The data manipulation language for MBDS, the attribute-based data language (ABDL) is a non-procedural language which originally supports four primary database operations: RETRIEVE, INSERT, DELETE and UPDATE. It is the purpose of this thesis to design and implement the fifth primary database operation, the RETRIEVE-COMMON operation.

The RETRIEVE request is used to retrieve records of the database. The syntax of a RETRIEVE request is shown as below:

RETRIEVE Query (Target-List) [BY Attribute] [WITH Pointer]

The query specifies which records are to be retrieved. The target-list is a list of output attributes. It may also consist of an aggregate operators on one or more output

attributes. MBDS supports five aggregation operators, they are: AVG, COUNT, SUM, MIN and MAX. The BY-clause and the WITH-clause are optional. The BY-clause may be used to group records when an aggregate operation is specified. The WITH-clause may be used to specify whether pointers to the retrieved records must be returned to the user or user program for later use in an update request. Some examples of retrieve request are shown in below.

Example 1. Retrieve the names of all employees who work in the Toy department.

```
RETRIEVE ((FILE=Employee) and (DEPT=Toy)) (NAME)
```

Example 2. List the average salary of all departments.

```
RETRIEVE (FILE=Employee) (AVG(SALARY)) BY DEPT.
```

The INSERT request is used to insert a record into the database. The syntax of as INSERT request is:

```
INSERT Record
```

The following example will insert a record into the Employee file.

```
INSERT (<FILE,Employee>, <SALARY,30000>, <DEPT, Toy>)
```

The syntax of a DELETE request is:

```
DELETE Query
```

where the query specifies the record(s) to be removed from the database. The following example will delete records from the Employee file.

```
DELETE ((FILE=Employee) and (SALARY=30000) and (DEPT= Toy)).
```

The UPDATE request is used to modify records of the database. The syntax of the UPDATE request is:

UPDATE Query <Modifier>

where the query specifies the particular records to be updated from the database and the modifier specifies the kinds of modification that need to be done on records that satisfy the query. The following example will give a $1000 raise to all employees.

UPDATE (FILE=Employee) <SALARY=SALARY+1000>

The RETRIEVE-COMMON request is used to merge two files by common attributes. It will be detailly discussed in the later chapters.

D. THE PROCESS STRUCTURE

MBDS is a message-oriented system. In a message-oriented system, each process corresponds to one system function. These processes communicate among themselves by passing messages. The processes are created at system start time and exist until the system is stopped. Figure 2.2 provides an overview of MBDS process structure.

1. The Communication Processes

Communication between computers in MBDS is achieved by using the PCL. MBDS provides a software abstraction to this bus for each computer in order to emulate broadcast capabilities. The abstraction consists of two complimentary processes. The first process, get-pcl, gets message from other computers off the PCL. The second process, put-pcl, puts messages on the bus to be broadcasted to other computers. Every computer, whether it is the controller or a backend, has its own get-pcl and put-pcl.

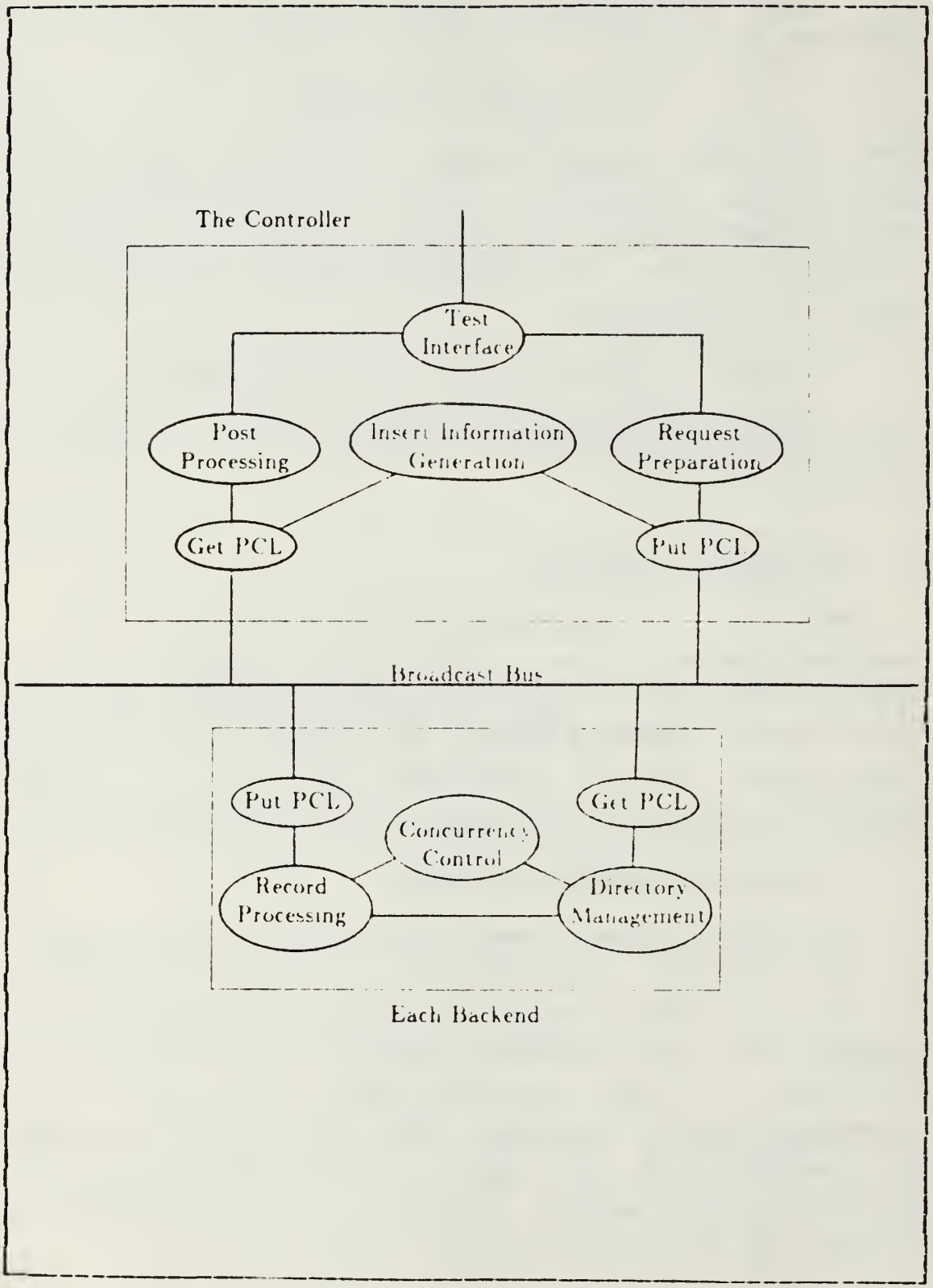


Figure 2.2 The MBDS Process Structure.

There are 31 message types and one general message format used in the MBDS message-passing facilities. The format (shown in Figure 2.3) is used for each of the three message-passing facilities, namely, messages within the controller, messages within the backends, and messages between computers.

A Message	Data Type
Message Type	a numeric code
Message Sender	a numeric code
Message Receiver	a numeric code
Message Text	an alphanumeric field terminated by an end of message marker

Figure 2.3 The General Format of MBDS Messages.

Messages between computers are divided into two classes: messages between backends and messages between the controller and the backends. Figure 2.4 describes each of MBDS message types.

2. The Test Interface Process

The test interface process allows the user to interact with the MBDS directly. Since MBDS does not use a host computer, the test interface process is contained in the controller.

3. The Processes of the Controller

In addition to the communications and test-interface processes, the controller consists of three additional processes: Request Preparation (RP), Insert Information Generation (IIG) and Post processing (PP). RP receives,

parses and formates a request (transaction) before sending the formated request (transaction) to the directory-management process in each backend. IIG is used to provide additional information to the backends when an insert request is received. PP is used to collect all the results of a request (transaction) and forward the results to the user.

4. The Processes of Each Backend

In addition to the communication processes, each backend also consists of three other processes: Record Processing (RP), Directory Management (DM) and Concurrency Control (CC).

DM controls the execution of a request at a backend and accesses the secondary-storage-based directory tables. It determines the disk addresses where the relevant data of a particular request are stored and then sends those disk addresses to RP.

CC is used to insure the consistency of the database while allowing concurrent execution of multiple requests.

RP performs the disk I/O operations and other operations specified by the request. It receives the secondary-addresses from DM, which processes the request. The results are then forwarded to the controller.

	MESSAGE-TYPE NUMBER AND NAME	SRC	DEST	PATH
1	TRAFFIC UNIT	1 HOST	1 REQP	1 HC 1
	REQUEST RESULTS	PP	HOST	CH
	NUMBER OF REQUESTS IN A TRANSACTION	REQP	PP	C
	AGGREGATE OPERATORS	REQP	PP	C
5	REQUESTS WITH ERRORS	5 REQP	5 PP	5 C 5
	PARSED TRAFFIC UNIT	REQP	DM	CB
	NEW DESCRIPTOR ID	IIG	DM	CB
	BACKEND NUMBER	IIG	DM	CB
	CLUSTER ID	DM	IIG	BC
10	REQUEST FOR NEW DESCRIPTOR ID	10 DM	10 IIG	10 BC 10
	BACKEND RESULTS FOR A REQUEST	RECP	PP	BC
	BACKEND AGGREGATE OPERATOR RESULTS	RECP	PP	BC
	RECORD THAT HAS CHANGED CLUSTER	RECP	REQP	BC
	RESULTS OF A RETRIEVE OR FETCH CAUSED BY AN UPDATE	RECP	REQP	BC
15	DESCRIPTOR IDS	15 DM	15 DMs	15 BB 15
	REQUEST AND DISK ADDRESSES	DM	RECP	B
	CHANGED CLUSTER RESPONSE	DM	RECP	B
	FETCH	DM	RECP	B
	OLD AND NEW VALUES OF ATTRIBUTE BEING MODIFIED	RECP	DM	B
20	TYPE-C ATTRIBUTES FOR A TRAFFIC UNIT	20 DM	20 CC	20 B
	DESC-ID GROUPS FOR A TRAFFIC UNIT	DM	CC	B
	CLUSTER IDS FOR A TRAFFIC UNIT	DM	CC	B
	RELEASE ATTRIBUTE	DM	CC	B
	RELEASE ALL ATTRIBUTES FOR AN INSERT	DM	CC	B
25	RELEASE DESCRIPTOR-ID GROUPS	25 DM	25 CC	25 B
	ATTRIBUTE LOCKED	CC	DM	B
	DESCRIPTOR-ID GROUPS LOCKED	CC	DM	B
	CLUSTER IDS LOCKED	CC	DM	B
29	NO MORE GENERATED INSERTS	RECP	REQP	BC
29	NO MORE GENERATED INSERTS	REQP	DM	CB
29	NO MORE GENERATED INSERTS	DM	RECP	BC
30	REQUEST ID OF A FINISHED REQUEST	30 RECP	30 CC	30 B 30
31	AN UPDATE REQUEST HAS FINISHED	RECP	DM	B
31	AN UPDATE REQUEST HAS FINISHED	DM	CC	B

SOURCE OR DESTINATION DESIGNATION	PATH DESIGNATION
HOST : HOST MACHINE (TEST-INT)	H : HOST
REQP : REQUEST PREPARATION	C : CONTROLLER
IIG : INSERT INFORMATION GENERATION	C : CONTROLLER
PP : POST PROCESSING	C : CONTROLLER
DM : DIRECTORY MANAGEMENT	B : A BACKEND
RECP : RECORD PROCESSING	B : A BACKEND
CC : CONCURRENCY CONTROL	B : A BACKEND

Figure 2.4 The MBDS Message Types.

III. DESIGN AND ANALYSIS OF THE RETRIEVE-COMMON REQUEST

In this chapter, we introduce the terminology and notaticns of the "Retrieve-Common" request, investigate and analyze several pcssible design and implementation approaches, and then select the best one to design and implement the Retrieve-Common operation for MBDS. The selection of an approach is based on the design requirements and the design issues of MBDS.

A. THE INTENDED OPERATION

1. An Operation Cn Two Files

The **RETRIEVE-COMMON** request is used to merge two files by common attribute values. The common attribute values are the attribute values which belong to the records of both files. For example, suppose there are two files: file A and file B. File A contains the records of the street names of San Jcse city:

(<FILE, A>, <STREET, MONTEREY>, <CITY, SAN JOSE>)
(<FILE, A>, <STREET, SECOND>, <CITY, SAN JOSE>)
.
.
.

File B consists the records of city names of the Monterey county:

(<FILE, B>, <CITY, MONTEREY>, <COUNTY, MONTEREY>)
(<FILE, B>, <CITY, SEASIDE>, <COUNTY, MONTEREY>)
.
.
.

The RETRIEVE-COMMON request can provide us a third file, say, file C, with the information such as: "All the records of both files A and B, where the street name of the records in file A is identical to the city name of the records in file B. One of the records in file C which satisfy the request would be

(<FILE, C>, <FILE, A>, <STREET, MONTEREY>, <CITY, SAN JOSE>, <FILE, B>, <CITY, MONTEREY}, <COUNTY, MONTEREY>).

Logically, the retrieve-common request involves two retrieval operations. We define the first retrieval operation as the source retrieve and the second retrieval operation as the target retrieve. The set of all the records that belong to the result of the source retrieve is called the source record set. The set of all the records that belong to the result of the target retrieve is called the target record set. A source (target) record is the record that belongs to the source (target) record set. Similarly, those attributes will be refered as source attributes and target attributes. The merged source and target records are termed the result record set. The aforementioned file C is a result record set.

We term the source and target attribute names that participate in the retrieve-common operation the join attribute names or briefly join attributes. However, their values are termed common attribute values, or simply common values. The retrieve-common operation requires that the join attribute which is specified in the source record set must have the same domain as that of the join attribute in the target record set, although they need not have the same attribute name.

Consider another example, suppose the source records are characterized by the attributes, Employee_name, Wages, and the target records are characterized by Rank, Wages.

Further, let the domain of the Employee_name be the character string and the domain of both Rank and Wages be the integer. A retrieve-common operation may be performed by merging on the attribute values of the wage of the respective source record and the target record. A retrieve-common operation may also be performed by merging on the wages of the source record and the ranks of the target record. Since their value domains are the same. However, a merge between the employee names and the ranks would not be permitted, since their domains are different.

The logical operation for the retrieve-common request can be described as follows.

(1) All records satisfying the source retrieve are collected.
(2) All records satisfying the target retrieve are collected.
(3) The records of the two collections are pairwise merged on the common (source and therefore target) attribute values.

2. The Syntax Of Retrieve-Common Request

When developing the syntax of the retrieve-common request, we must attempt to design a data language construct that is similar, syntactically, to the other primary operations of ABDI. In particular, the syntax of retrieve-common operation should resemble the syntax of the ABDL retrieve operation given below:

RETRIEVE Query (Target-list) [BY Attribute] [WITH Pointer]

Using the above syntax as a guideline, we define the syntax for the retrieve-common request as follows.

RETRIEVE Query-1 (Target-list-1)[BY Attribute][WITH Pointer]
COMMON (Attribute-1, Attribute-2)

RETRIEVE Query-2 (Target-list-2)[BY Attribute][WITH Pointer]

The retrieve-common request consists of three parts. The first part is what we have referred to as the source retrieve request, which retrieves the source record set. The second part is the specification of the join attributes, where Attribute-1 belongs to the source record and Attribute-2 belongs to the target record. Although the values of these two attributes must be the same in order to satisfy the condition for merging the respective records, their attribute names need not be identical. The third part is what has been refered to as the target retrieve request, which retrieves the target record set.

B. AN ANALYSIS OF DIFFERENT DESIGNS

In order to make this thesis self-contained, several possible design approaches described in [Ref. 8] are reviewed in this section.

The main issue when considering alternative strategies for implementing the retrieve-common request is where the merge of the source and the target records should be performed.

There are three major alternatives for distributing the workload of the retrieve-common request.

(1) The controller does all of the merge operation.
(2) The backends do all of the merge operation.
(3) The controller and the backends share the workload of the merge.

Each of these alternatives will be analyzed and judged using the design requirements and design issues of MBDS.

In order to simplify the analysis of design (or implementation) strategies, we make the following assumptions.

(1) The records of the source record set and the records of the target record set are distributed evenly across the backends.

(2) The operation of the retrieve-common is performed as described in the previous section.

1. The Controller Does All the Merge Operation

In this alternative, each backend only performs these two retrieval operations and then sends the records of source record set and records of the target record set to the controller. Upon receiving all the source records and target records from all the backends, the controller performs the merging operation and sends the results to the host computer.

2. The Controller And The Backends Share The Merge Operation

Each backend performs the merge operation over its source records and target records. The merged records, along with the source and target record sets are then sent to the controller. The controller performs the merge operation over the source and target record sets coming from different backends and then sends the results together with the previously merged records (done by individule backends) to the host.

3. The Backends Do All the Merge Operation

This alternative may be further broken into two subalternatives.

(a) The backends share the merge operation.
The backends send either source or target records to each other. Let's assume that the target records are sent. Each backend will have a portion of the source record set and a whole set of target records. Then,

the backends perform the merge operation over its own source records and all of the target records, and sends the results to the controller.

(b) One designated backend performs the merge operation. All records of both the source record set and the target record set are sent to the designated backend from all of the other backends. The designated backend performs the entire merge operation and sends the results to the controller.

4. An Analysis of the Design Approaches

Four alternatives of distributing the workload of the merge operation among the controller and the backends have been discussed in previous subsection. We now examine these alternatives with the design goals of MBDS.

Alternative 1, where the controller performs the entire merge operation will increase the workload of the controller. Recall that in chapter II we have stressed that in order to reduce the chance of the controller being the bottleneck of the system, we minimize the work of the controller. Alternative 1 violates this design requirement Therefore, it will not be considered further.

Alternative 2 will increase the communications load and increase the workload of the controller. This alternative complicates the first and the sixth design issues of MBDS. Therefore, it will also be eliminated from the design consideration.

Alternative 3a meets the design issue of minimizing the controller function and distributing the workload to each backend evenly. Alternative 3b does not increase the workload of the controller; nor does it distribute the workload to each backend. Furthermore, transmitting all the records of both the source record set and target record set

will increase the communications overhead. In addition, performing the entire merge operation in one backend will unbalance the workload, thereby reducing the parallelism of the backends, i.e., by having a single-backend to do the merge and all other backends to idle. This complicates both of the third and sixth design issues, so this alternative is also eliminated.

With this analysis we choose the alternative 3a as our design approach. That is, **each backend performs a partial merge with its portion of source records and all target records. And then, sends its result to the controller. The controller forwards the final result to the host computer.**

C. AN ANALYSIS OF DIFFERENT IMPLEMENTATIONS

Three different implementations for merging the source and the target record sets are considered.

(1) A straightforward implementation.
(2) An implementation based on sorting and matching.
(3) An implementation based on bucket-hashing.

1. The Straightforward Implementation

The concept of this alternative is very simple and the merging operation is based on the "nest-loop" algorithm [Ref. 8 : p. 86] which is shown in Figure 3.1.

This alternative is accomplished in five phases:

(1) Each backend determines its own source records and stores them into a predefined portion of the secondary storage area.
(2) Each backend determines its own target records and stores them into the predefined portion of the secondary storage area.

```
PROCEDURE Nest_loop_merge
   FOR each record in the source record set DO
      FOR each record in the target record set DO
        IF the merging condition is satisfied
          THEN
             form a result record
        END IF
      END FOR
   END FOR
END PROCEDURE Nest_loop_merge
```

Figure 3.1 The Nest-loop Merge Procedure.

(3) Each backend broadcasts its own local target records to all of the other backends.

(4) Each backend receives the broadcasted target records from the other backends and stores them into the secondary storage together with its own target records.

(5) Each backend brings its own source records and the entire target record set into the primary memory, performs the "nest-loop" merging operation and then send the merged results to the controller.

2. The Implementation Based on Sorting and Matching

The idea of this implementation is based on the following inference.

Since the retrieve-common operation is simply a merging operation on two files of records sets, if we can have these two files presorted by the values of their common attributes then the merging operation may be efficiently

performed by matching the values of the common attributes of the records of these two files.

There are two possible alternatives to perform the sort-match algorithm.

(a) The backends do all of the sorting and matching operations.
(b) The backends and the controller share the sorting and matching operations.

Alternative (b) will increase the workload of the controller and contradict with the design goals of MBDS, and is therefore eliminated from consideration. Only alternative (a) will be examined. Alternative (a) accomplishes the retrieve-common operation in four phases.

(1) Each backend retrieve, sorts and stores its own source records and target records separately, and then broadcasts either set of records to the other backends. (Let's assume that the target records are transmitted.)
(2) Each backend receives and merges the incoming non-local target records into its own local target records.
(3) Each backend performs the matching operation over its own portion of source records and the entire set of target records (from all the backends).
(4) The backends send the results to the controller.

3. The Implementation Based on Bucket-Hashing

This implementation strategy attempts to speed up the comparison and merge by hashing records into small groups (the buckets of the hashing table) which contain records with common attribute values, so that the time complexity of the merging operation may be reduced.

A hashing function applied to the common attribute values is used to hash records into buckets. The bucket numbers are consecutive integers. Instead of using primary and overflow areas, the buckets use one or more fix-sized blocks to store records. The numbers of blocks may vary among buckets. Details of the hashing table, the buckets and the the blocks will be described in the next chapter.

Those source records and target records within the same bucket will be examined and merged if the merging condition is matched. This alternative can also be broken to two subalternatives.

(a) One common hashing table is used for both source and target record sets.

(b) Two separate tables are used, one for each record set.

a. One Common Hashing Table

This alternative is accomplished by each backend in four phases:

(1) All local source records will be hashed and stored into blocks according to their hashed values. These blocks (therefore buckets) are termed source blocks (buckets).

(2) After all the local source records have been hashed, the local target records are hashed one at a time and buffered. If the target record is hashed into an empty source bucket, then it is buffered for transmitting to other backends. Otherwise, all the records in the source bucket will be retrieved and merged with that target record only if the merging condition is satisfied. The results are first buffered and then sent to the controller.

(3) Since the non-local target records may arrive at a backend while the backend is processing some other records, each backend will place these incoming records on a predefined secondary storage area.

(4) Each backend retrieves the non-local target records from the secondary storage area and processes them in the same way as the the backend does on its local target records.

b. Separate Hashing Tables

This alternative is accomplished in three phases.

(1) The backends will hash and store their own source records and target records into two separate hashing tables by a common hashing function. After all of the target records have been hashed and stored, each backend will broadcast the hashed results of their target records (i.e., the bucket number and the records associated with that bucket number) to all of the other backends.

(2) Upon receiving all of the target information from the other backends, each backend stores those target records into appropriate buckets according to their bucket numbers.

(3) The backends perform the merge operation on the local source records and the entire set of target records and send the results to the controller. The procedure is shown in Figure 3.2.

4. A Comparison Of The Three Implementation Approaches

In this section we compare and analyze these implementation approaches. Since the backends work in parallel, our analysis only focuses on how much time it takes for one backend to do one particular strategy. There are common operations that each backend performs, so that the time complexities for these operations can be ignored when comparing the implementation strategies. The times of these common operations are:

```
PROCEDURE Hashing_merge
  FOR the bucket_value = min_value to max_value DO
     IF the buckets of both tables are not empty
        then
            retrieve all the records from both buckets
            perform merge operation based on
            the straightforward algorithm
     End IF
  END FOR
END PROCEDURE Hashing_merge
```

Figure 3.2 The Hashing_merge Procedure.

(1) the time to process the records for the source request which involes determining which records of the database satisfy the query, projecting the attribute-value pairs of the target-list of the satisfied records and forming a source record set;

(2) the time to process the records for the target request, which involes determining which records of the database satisfy the query, projecting the attribute-value pairs of the target-list of the satisfied records and forming a target record set;

(3) the time to broadcast the local target records to the other backends; and

(4) the time to send the merged results to the controller.

The following notions are introduced to simply the ensuing analysis.

Cs : Cardinality of the source record set in one backend.

Ct : Cardinality of the target record set in one backend.

Cb : Average number of records in a bucket.

M : Number of Backends.

B : Number of Index Entries in the hashing table.

Ti : Average time tc read (write) a block of records from (to) secondary storage.

Tb : Average time tc read (write) a record form (to) a bucket.

Tc : Average time to compare the common attribute values of two records.

Th : Average time tc hash a record.

Tm : Average time tc merge two records.

a. An Analysis for the Straightforward Implementation

We recall that there are five phases in this implementation as discussed in a previous section.

Phase 1: Since there are Cs local source records in each backend, the time complexity for storing them into the secondary storage is:

Ti*(Cs/Cb).

Phase 2: Since there are Ct local target records in each backend, the time complexity for storing them into secondary storage is:

Ti*(Ct/Cb).

Phase 3: The time complexity for this phase is ignored.

Phase 4: Since each backend receives (M-1)*Ct target records from the other backends, the time complexity for storing them in the secondary is:

(M-1)*(Ct/Cb)*Ti.

Phase 5: Records are merged in this phase. There are Cs source records and M*Ct target records in each backend. Each block of the source records is compared and merged with all of the target records. It takes Ti to bring one block of source records into the primary memory from the

secondary storage and M*(Ct/Cb)*Ti for the entire target record set.

It takes Cb*Tb to access one block of source records and M*Ct*Tb to access all of the target records. The time complexity for comparing one block of the source records and all of the target records is

$$Cb*M*Ct*Tc.$$

We further assume that there are k fraction of target records participating the merging operation. The time complexity for merging one block of source records and all of the target records becomes:

$$k*M*Ct*Tm.$$

The total time complexity for processing one block of source records of this implementation is:

$$[Ti+M*(Ct/Cb)]+(Cb+M*Ct*Cb)*Tb+(Cb*M*Ct*Tc)+(k*M*Ct*Tm).$$

There are Cs/Cb blocks of source records in each backend; therefore, the time complexity of this alternative is:

$$(Cs/Cb)*\{[Ti+M*(Ct/Cb)]+(Cb+M*Ct*Cb)*Tb + (Cb*M*Ct*Tc)+(k*M*Ct*Tm)\}$$

or

$$(M*Cs*Ct)*[Ti+(Tb+k*Tm)/Cb+Tc]+Ti*(Cs/Cb)+Cs*Tb$$

Because Cs may be equal to Ct and M is a small constant, the time complexity may be further simplified to be

$$O(Cs*Ct) \text{ or}$$

$$O(Cs^2).$$

b. An Analysis for the Sort-Matching Implementation

We will analyze each phase of this implementation approach.

Phase 1: Each backend sorts its two record sets and broadcasts the sorted target record set to the other

backends. Due to the large size of records, the sorting operation can not be done by using an internal sorting algorithm. There are several external sorting algorithms which can sort the local source records and the local target records with the time complexities of O(Cs*(logCs)) and O(Ct*(log Ct)), respectively. However, these algorithms all have some limitations: either using special hardware configuration or running different software among processors [Refs. 9,10].

Because we do not want to put limitations on the hardware configuration of MBDS and to use different software among the backends, this alternative is eliminated from our consideration.

c. An Analysis for the Bucket-Hashing Implementation

In order to further simplify our analysis, we assume that the local source records and target records can be evenly hashed across all the buckets of the hashing tables and each bucket will contain only one block of local source records or one block of local target records. First, we analyze the alternative that uses only one hashing table.

Phase 1: Each source record needs to be hashed, written into a bucket by its hashed value. This includes getting the block of that bucket from the secondary storage and writing the record into the block and returning the block to the secondary storage. Therefore, the time complexity for each backend to hash and store the source records is:

$$Cs*(Th + Tb + 2Ti).$$

Phase 2: Every time a target record is hashed, the bucket with that hashed value is checked. If the bucket is not empty, then all the source records in that bucket

will be retrieved into the primary memory, compared with the target record and merged with it if their common attribute values are equal. The time complexity for bring one bucket (block) of source records into primary memory is Ti. The time complexity for accessing those source records from the block and comparing with that target record is:

$$Cb * (Tb + Tc).$$

Suppose that the probability of hashing a target record into a non-empty bucket is p and the probability of satisfying the merging condition is f, then the time complexity for each backend to process one local target records is:

$$Th + p * [Ti + Cb * (Tb + f * Tc)].$$

Because we assume the source records are evenly hashed across the buckets of the hashing table, p is equal to 1. There are Ct local target records in each backend so that the time complexity for each backend to process its local target records is:

$$Ct*\{Th+[Ti+Cb*(Tb+Tc+f*Tm)]\}.$$

Phase 3: Each backend receives (M-1)*Ct target records from other backends. The time complexity for storing these records back to the secondary storage is:

$$(M-1)*(Ct/Cb)*Ti.$$

Phase 4: It takes (M-1)*(Ct/Cb) for each backend to retrieve all the non-local target records from the secondary storage into the primary memory. The time complexity for processing these records is:

$$(M-1)*Ct*\{Th+[Ti+Cb*(Tb+Tc+k*Tm)]\}.$$

The time complexity of this phase is:

$$(M-1)*(Ct/Cb)*Ti+M*Ct\{Th+[Ti+Cb(Tb+Tc+f*Tm)]\}.$$

The total time complexity of this alternative for a backend is:

$$Cs(Th+Tb+2Ti)+2(M-1)*(Ct/Cb)*Ti$$
$$+M*Ct\{Th+[Ti+Cb(Tb+Tc+f*Tm)]\}.$$

Now, we analyze the other alternative which uses two separate hashing tables.

Phase 1: The source records and the target records will be hashed, grouped into the buckets of separate hashing tables and then placed onto the secondary storage. The time complexity for each backend to process its local records is:

$$(Cs+Ct)*(Th+Tb+2Ti).$$

Upon receiving the target records from the other backends, each backend will insert those incoming records into the hashing table of the target records and stored them back to the secondary storage. Since those non-local target records are grouped and sent by their bucket numbers, the insertion time is so quick that it may be ignored. By using an inverted list, the time complexity for each backend to return those incoming target records to the secondary storage is:

$$(M-1)*(Ct/Cb)*Ti.$$

Phase 2: Records of these two hashing tables will be processed one bucket at a time. For any bucket number (i.e., a table entry), if the buckets of both hashing tables are not empty, then all blocks of the records of both buckets will be read into the primary memory for the merging operation. It takes Ti for bringing one bucket of source records (in this case, one block) into the primary memory and M*Ti for one bucket of target records (M blocks). The time complexity for accessing, comparing and possibly

merging one bucket of source records with one bucket (M blocks) of target records (not including the disk I/O time) will be:

$$Cb*[Ib+M*Cb*(Ib+Tc+f*Tm)].$$

The expected time complexity for all buckets will be:

$$(Cs/Cb)*Cb*[Tb+M*Cb*(Tb+Tc+f*Tm)]$$

Therefore, the total time complexity for this alternative is:

$$(Cs+Ct)(Th+Tb+2Ti)+(M-1)*(Ct/Cb)*Ti$$
$$+(Cs/Cb)*Cb*[Tb+M*Cb*(Tb+Tc+f*Tm)]$$

	One Common Table	Two Separate Table
Th	Cs+M*Ct	(Cs+Ct)
Tb	Cs+Ct*M*Cb	(M+2)*Cs+Ct
Tc	M*Ct*Cb	Cs*M*Cb
Ti	2Cs+M*Ct+2(M-1)*(Ct/Cb)	(Cs+Ct)+(M-1)*Ct/Cb
Tm	M*Ct*Cb*f	Cs*M*Cb*f

Figure 3.3 The Time Complexities of the Bucket-Hashing Implementations.

A summary of the time complexity in terms of Th, Ti, Tb, and Tc for these two subalternatives is shown in Figure 3.3. As shown in Figure 3.3, alternative which uses two separate tables is better than the other one which

employs only one table. Since Cb and M are constants, f is smaller than 1 and Ct may be equal to Cs, we can further simplify the the time complexity of the two-separate-tables subalternative to be:

$$O(Cs+Ct) \text{ or}$$

$$O(Cs).$$

d. The Conclusion for Our Implementation Approach

A summary of the analysis for those implementation approaches in terms of time complexity are shown in Figure 3.4. Clearly, the one based on Bucket-Hashing with two separate hashing tables is the best approach. Therefore, our implementation will be based on that approach. The details of design and implementation will be discussed in the next chapter.

Straightforward	$O(Cs^2)$
Sorting-Matching	Not considered
Bucket Hashing	O(Cs)

Figure 3.4 Time Complexity of Different Implementation.

IV. DETAILED DESIGN FOR IMPLEMENTING RETRIEVE-COMMON OPERATION INTO MBDS

In the previous chapter, a bucket-hashing based implementation approach has been selected for implementing the retrieve-common operation into MBDS. In this chapter, we focus on specifying the details of that approach and discuss any of the existing MBDS software which will be affected by this implementation. Our primary goal is to use the existing software as much as possible and to minimize the effects which may be caused by the implementation.

The operations of the retrieve-common request may be described in four phases. First, the user's request must be preprocessed so that all backends can be informed by an appropriate message. This is the request-preprocessing phase. Second, the records of both the source and the target record sets are retrieved before the merging operation. This is the record-retrieving phase. Third, those retrieved records are hashed on the values of their join attributes and stored into a hashing table according to their hashed values (i.e., the bucket numbers). We recall that there are two hashing tables, one for the source records and one for target records. Further, the hashed local target records are broadcasted to the other backends. This is the hashing-and-storing phase. Lastly, hashed records of source buckets and hashed records of target buckets are compared and merged bucket-by-bucket, respectively. The merged results are sent to the controller from all of the backends. This is the merging phase. The controller then forwards those results to the host computer.

The operations of the first and second phases can be done by the existing system software with minor

mo ications. However, in order to accomplish the ope tions of the last two phases, we must design a new set of procedures, which we have referred to as the hashing module. In the remainder of this chapter, we first describe the hashing module, and then the operations of those four phases.

A. THE HASHING MODULE

This module is designed to accomplish the operations of the last two phases of the retrieve-common request. There are three procedures within this module. They are: the hashing procedure, the bucket-block tracking procedure and the merging procedure. In this section, we first discuss the two different alternatives for implementing this module. After choosing the better alternative, we then describe the three procedures of the hashing module.

1. Alternatives for Implementing the Hashing Module

There are two alternatives that may be used for implementing the hashing module. In the first alternative, the hashing module is implemented as a separate process of the backend. This alternatives modifies the existing process structure of a backend by introducing a sixth process and its associated communication paths into each backend. In the second alternative, the hashing module is implemented as part of the existing record processing process (RECP). This alternative leaves the existing backend process structure unchanged.

a. As a Separate Process

In this alternative, the hashing module is designed as a separate process of the backend. The inputs to the hashing module are either the local source or target

records from the local RECP or the other target records from the RECPs of the other backends. The outputs from the hashing module are the merged results, which are sent to the controller. The transfer of records between processes (i.e., non-local target records from "Put Pcl" to the hashing module or the local source records or the local target records from the local RECP to the hashing module) is accomplished using the interprocess message capabilities of each backend. The new process structure of each backend with the additional communication paths is shown as Fig 4.1. Since the hashing module is an independent process, the effects of this implementation on the other processes of MEDS may be minimized.

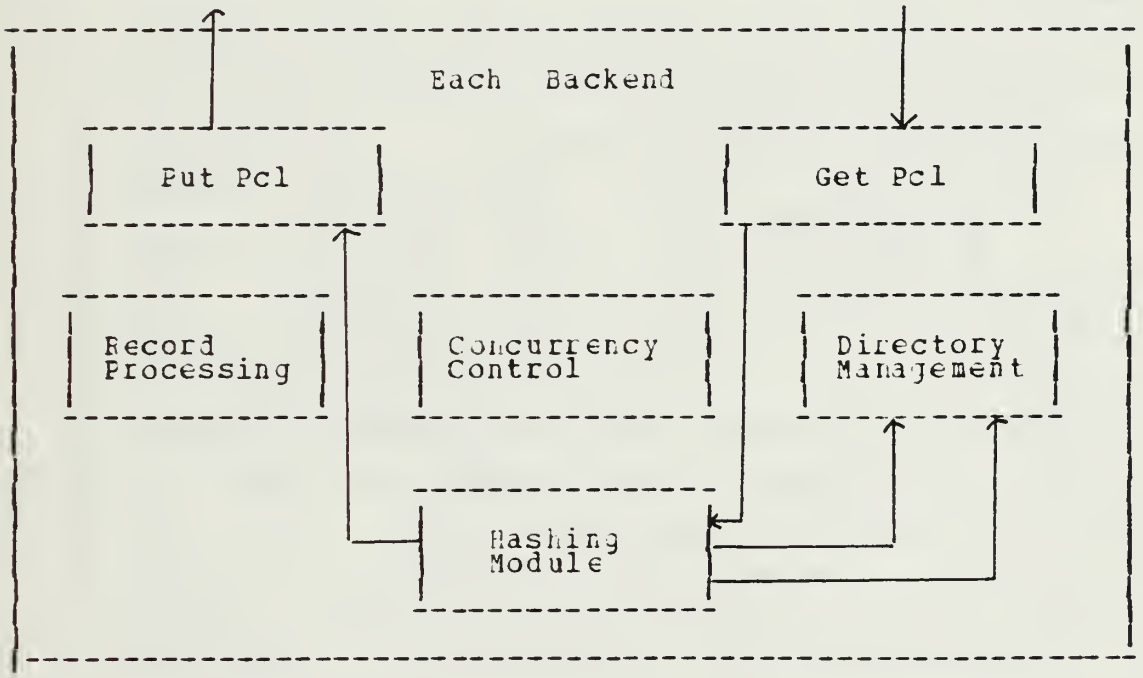


Figure 4.1 Hashing Module As a Separate Process.

b. As a Procedure within Record Processing

In this alternative, the hashing module is designed as a group of procedures that are added to RECP. In Figure 4.2 we show the structure of the hashing module with RECP. The local records (both the source records and the target records) are retrieved by the physical data operation of RECP of each backend. Once the records are retrieved, they are sent to the hashing module. The non-local target records are received by RECP from the other backends and then passed to the hashing module. The merged results are then sent to the controller. With modularized programming, the hashing module may be independently implemented with a minimal effect on the original RECP software.

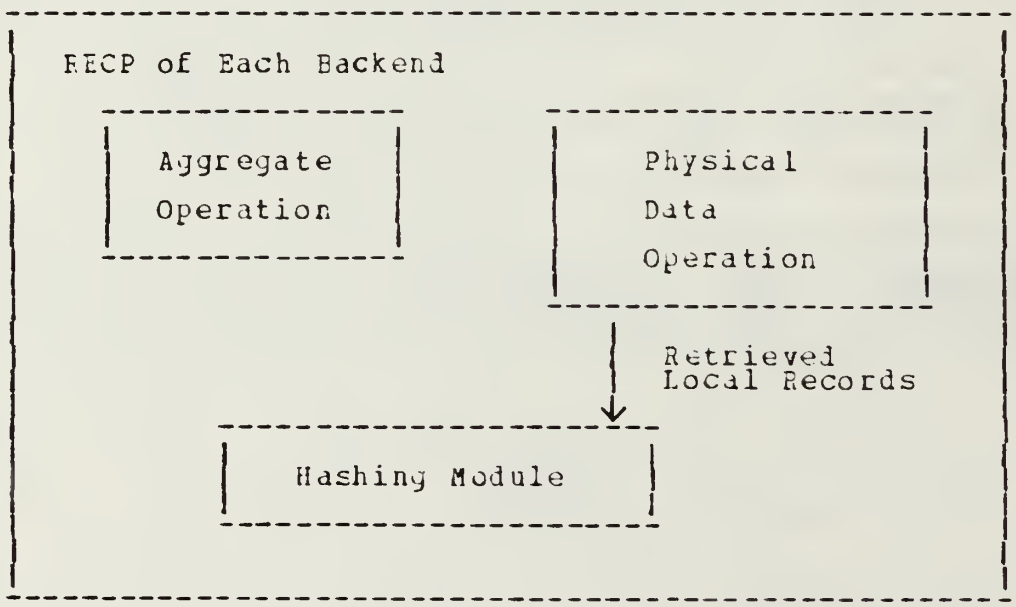


Figure 4.2 Hasing Module as Part of RECP.

c. Comparison of These Two Alternatives

Both alternatives can be easily implemented with minimal effect on the existing system. The difference between these two alternatives is the way that the local records are passed from the "physical data operation" to the hashing module. In alternative (a), the records are passed as an interprocess message. In alternative (b), the records are passed as a parameter of a procedure call. We choose alternative (b) for three reasons.

(1) The message-passing between two processes within a backend is slower than the parameter-passing. In message-passing, both processes have to access a common memory to put (or get) message. The accessing time coupled with the time required to place a message in the common memory by the sender and fetch the message from the common memory by the receiver is considerable. In parameter-passing, only the logical address of the record buffer is passed between the procedures, which is much simpler and faster.

(2) Even if message-passing within a computer is extremely fast, there is a large number of messages (i.e., records) which is considerable. Since it amounts to route the messages (records) between two processes.

(3) The extra communication paths required by alternative (a) (i.e., the communication paths among the hashing module and the other MBDS processes), increase the number of messages passed within a backend and among backends. By increasing the inter-backend and intra-backend communication, we may adversely effect the overall performance of a backend.

2. The Hashing Procedure

This procedure is used to perform the hashing operation on the values of the join attributes of the input records. The inputs to the procedure are either the local source records or the local target records, which are received from the physical-data-operation subprocess of RECP. The output from the procedure are the input records and their hashed values (i.e., the bucket numbers), which are sent to the bucket-block tracking procedure with the request id for further processing.

The hashing operation is done by the hashing functions of this procedure. Since the type of the values of the join attributes may either be an integer or a character string, we have designed two hashing functions in this procedure. Generally, a good hashing function should satisfy the following three requirements:

(1) All of the records should be evenly distributed into buckets of the hashing table;
(2) The chance of hashing different records into the same bucket should be minimized; and
(3) The hashing computation should be fast.

These requirements are closely related to the number of buckets and the hashing algorithm which is used in the hashing function.

a. The Number of the Buckets

A hashing table with a large number of buckets is useful for a number of reasons. First, the large number of buckets may reduce the chance of hashing different records into the same buckets. Second, the number of records in each bucket is also quite small, and this will reduce the access time during merging. However, it would be

impractical to have a table with a very large number of bucket entries, where each bucket would only contain a few records. When the table becomes exceedingly large, a substantial cost is incurred to maintain the bucket index. The bucket index of a hashing table is an array of fixed-size bucket entries. There is a bucket entry for each bucket to keep track of the records which are stored in that bucket. Therefore, the number of buckets (and therefore the bucket entries) can be computed by the following equation:

Let X be the size of the bucket index (measured in bytes),
Y be the size of a bucket entry (measured in bytes),
then the number of buckets is (X / Y).

For example, if the size of bucket index of a hashing table is 8K bytes and the size of each bucket entry is 8 bytes then the number of bucket entries for that hashing table is 1k, i.e., 1024.

How should we determine the size of the bucket index of our hashing table? Since MBDS allows the concurrent execution of different user transactions, there may be a number of retrieve-common requests being processed by the system. Each of the retrieve-common requests requires two hashing tables, one table for the source record set and one table for the target record set. Because of the potentially large number of hashing tables concurrently in use, it will be necessary to store the bucket indexes of the tables in the secondary storage and stage them into the primary memory on demand. To minimize and optimize the size of the bucket index of the hashing table, it is desirable to have the size of the bucket index as a multiple of the unit of disk I/O transfer. For example, if the unit of disk I/O transfer (which is typical the track size) is 4K bytes, then the size of the bucket index shall be M*4K bytes, where M = {1, 2, 3, ...}. In our case, we choose 16K bytes to be the

size of our hashing table, yielding 2048 entries (therefore, 2048 buckets) in the hashing table each with a bucket entry size of 8 bytes.

b. The Hashing Algorithm

Since the value type of the join attribute may be either an integer or a character string, we have designed two hashing functions, one for each value type.

(1) The Hashing Algorithm for the Integer-Valued Attributes. In order to evenly distribure the values of all join attributes into the buckets and to minimize the collisions; we use the information about the maximum and minimum values of the join attributes. These information is maintained in the record templates. The hashing algorithm for the integer attribute value is described as follows.

```
Step 1: Get the MAX (maximum) and MIN (minimum) values of
        the join attribute from the record template. Let
         X = The_number_of buckets_in_hashing_table
Step 2: If MAX-MIN ≤ X
           then  go to step 4
           else  Temp1 = (MAX - MIN) Div X
Step 3: Get the input record and let
         Y = The_value_of_the_join_attribute
         bucket_number = (Y - MIN) Div Temp1
         go to step 5
Step 4: Get the input record and let
         Y = The_value_of_the_join_attribute
         bucket_number = Y - MIN
Step 5: Return the bucket number to the calling procedure.
```

(2) The Hashing Algorithm for the Character-Valued Attributes. The record template does not

The record template does not provide the maximum and the minimum values for the character-valued attributes as it does for integer-valued attributes. In order to minimize collisions and distribute records evenly into buckets, we design a lookup table, which is an array with 2048 character-string elements, to perform the hashing function. The number of the elements is equal to the number of the entries in the bucket index of the hashing table. The values of the join attributes of the input records are searched against the contents of the lookup table to obtain the bucket values. The binary search algorithm is used to minimize the searching time of the lookup table.

The contents of the entries of the lookup table are created in the following way:

(1) Get a English dictionary with more than 2048 pages;
(2) Divide the page number by the number of the buckets (in our case the number is 2048);
(3) Let the result be x.y, where the x and y are positive decimal digits;
(4) Pick up the last word of every x.y page from the dictionary and place the first four characters as an entry in the lookup table; and
(5) If the length of the selected word is less than 4, fill the word with trailing blanks.

We use only the first four characters to compare the values of join attributes for two reasons. First, we believe that there are very few English words that will have the same first four letters. Second, we want to reduce the primary-memory requirements for the lookup table.

The algorithm for the character-valued attributes is as follows.

Step 1: Let MIN = 0 and MAX = 2047.
Step 2: Get the input record and let
X = The_value_of_the_join_attribute;
Step 3: If X ≥ look_up_table[MAX]

then

bucket_number = MAX, go to step 6.

Step 4: Use binary search to find the bucket number.

Step 5: Return the bucket number to the calling procedure.

3. The Bucket-Block Tracking Procedure

The input to this procedure may be either the local records (either the source records or the target records) with their bucket numbers from the hashing procedure or the non-local target records grouped by their bucket values from the other backends. The outputs from the procedure are the logical addresses of the hashing tables of the source request and the target request, which are sent to the merging procedure for the merging operation. The bucket-block tracking procedure performs three functions:

(1) maintaining a global table to keep track of the logical addresses of the hashing tables for all retrieve-common requests which are currently being processed in the system;

(2) maintaining a hashing table for the current request and keeps track of all of the buckets and blocks of that hashing table; and

(3) storing the input records into appropriate buckets and blocks according to their bucket values.

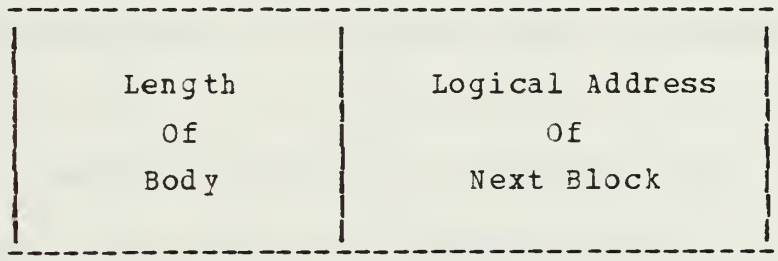
In order to provide a better understanding of this procedure, we first introduce the structures of the blocks, the buckets, the hashing table and the global table. We then discuss how these functions are accomplished.

a. The Structure of a Block

Each block is divided into two parts: the header and the body. The header has two fields. The first field is used to record the length (in bytes) of the body, i.e., all

of the records in bytes currently stored in this block. The second field is used to store the logical address of the next block whose records have the same bucket value as this block. If there is no other block of the bucket, then there is a null address in this field. The body is used to store the hashed records and their common attribute values. Blocks which are in the same bucket are maintained as an inverted list and tracked by their logical addresses. The structures of the block and its header are shown in Figure 4.3.

B. The Structure of a Block



B. The Structure of Block Header

Figure 4.3 The structures of Block and Its Header.

b. The Structure of a Bucket

As mentioned in chapter II, instead of using primary and overflow areas, each bucket uses fixed-size blocks to store records. The number of blocks per bucket may vary among different buckets. The bucket entry is used to indicate the status and to keep track of the blocks of that bucket.

Each bucket entry in the bucket index has two parts: the status and the logical address of the block currently being used. The status is used to indicate whether or not the bucket is empty. The size of the bucket entry is 8 bytes, where 2 bytes are used for the status and 6 bytes are used for the logical address which is represented by a tuple consisting of the logical disk number, the logical cylinder number and the logical track number. The structure of a bucket is shown in Figure 4.4.

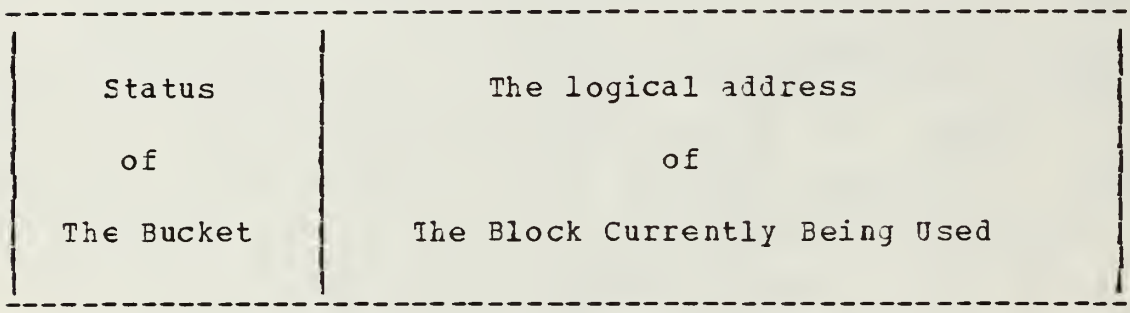


Figure 4.4 The Structure of a Bucket-entry.

c. The Structure of the Hashing Table

A hashing table is an array of bucket entries. We anticipate that the retrieve-common operation will be implemented on a SUN Workstation running the UNIX operating system, with a 16K unit of disk I/O. Using the equation from the previous subsection, we can compute the number of bucket entries for our hashing table to be 2048.

d. The Global Table

Since MBDS allows concurrent processing during the retrieval operation, there may be several retrieve-common requests in the system. We need a table that keeps track of all of the logical addresses of the

hashing tables for each retrieve-common request. Each entry of the global table contains two parts: the request id of the request and the logical address of the hashing table for that request. The request id consists of the traffic id, which is the unique identifier of a traffic unit [Ref. 11 : p. 41], and the request number which indicates the sequence of the request in the traffic unit. Each entry of the global table is created whenever a new hashing table is created, and deleted when that request has been completed processing. The structure of the global table is shown in Figure 4.5.

Request ID		Logical Address of Hashing Tables
Traffic ID	Request No	
:	:	:

Figure 4.5 The Structure of the Global Table.

e. The Sequence of the Operations of the Bucket-block Tracking Procedure

The steps of the sequence to accomplish the operations of this procedure are described as follows.

Step 1: Create and initialize the global table.

Step 2: Check the request ID of the input records with the global table to see if the input records belong to a new request. If they do, then allocate a hashing table for that request, initialize the bucket index and store the logical address of the hashing table into the global table. Otherwise, get the existing hashing table into the primary memory using the logical address information provided by the global table.

Step 3: Extract a record from the input buffer. If the record is the first record of that request, then go to step 10.

Step 4: If the bucket value of this record is the same as the previous one, then go to step 8.

Step 5: Store the block which contains the previous record back to the secondary storage.

Step 6: Get the desired bucket entry (table entry) for the record by its hashed bucket-value. Check the status of the bucket. If it is "empty", then go to step 11.

Step 7: Get the currently used block by its logical address in the bucket entry.

Step 8: If there is space in the block that is available for storing this record, then go to step 12.

Step 9: Get a new block, put the current logical address of the bucket entry into the "logical address of next block" field of the block header. Then, update the bucket entry with the logical address of this new block. Goto step 12.

Step 10: Get the desired bucket entry by its hashed bucket-value, update the status of that bucket entry to "not empty".

Step 11: Get a new block and put its logical address into the bucket entry.

Step 12: Store the record into the block and update the "length of record" field of the block header.

Step 13: Repeat the steps 3 to 12 until all records have been processed.

Notice that the block is not immediately returned to the secondary storage after the insertion of one input record. Since the records in MBDS are stored by clusters, it is very likely that records within the same cluster will be retrieved again. Therefore, by keeping the current block in the primary memory, we may save one store and one read operations if the next input record is retrieved from the same cluster and hashed into the same bucket (that is, they may have the same bucket value).

4. The Merging Procedure

This procedure is used to perform the merging operation. The inputs to this procedure are the logical addresses of the hashing tables of the source request and the target request, which come from the bucket-block tracking procedure. The outputs from this procedure are the merged results, which are sent to the controller.

The algorithm of the merging procedure is as follows.

Step 1: Reserve a result buffer.

Step 2: Get the hashing tables of the source request and the target request by their logical addresses.

Step 3: Compare the bucket statuses of these two hashing tables bucket by bucket. If both buckets contain records for a particular bucket number, then retrieve all the records associated with this particular bucket value from both tables.

Step 4: Apply the straightforward merging algorithm on those retrieved records. Insert merged results into the result buffer.

Step 5: If the result buffer is full, then send its contents to the controller.

Step 6: Repeat steps 3, 4 and 5 until all the buckets have been processed.

Step 7: Free the result buffer.

B. THE OPERATIONS OF THE FOUR PHASES

In this section we discuss the operations of each phase of the retrieve-common request and the software which will be affected by those operations.

1. The Request-preprocessing Phase

a. The Operations

The operations of this phase include parsing the user's transaction (or request) and if the transaction (request) is correctly parsed, then the controller will compose an appropriate message to inform the backends to begin execution for the request. Since the retrieve-common request is conceptualized and executed as two retrieval operations, the parser has to parse the user's request and transform the request from the form of a single request to a form of a transaction with two requests.

b. The Affected Software

Basically, operations of this phase can be done by the existing Request Preparation process. However, the software for this process must be modified as follows:

(1) The parser should be able to recognize the newly added syntax and correctly parse the request;

(2) The composer should be able to form a new message to inform PP and all of the backends so that they can perform the desired operation;

(3) New message types are added for processing the retrieve-common request; and

(4) PP and all of the backends should be able to recognize and process the new created message for the retrieve-common request.

2. The Record-retrieving Phase

a. The Operations

Operations of this phase include the address generation and the record retrieval for both the source request and the target request. These two requests will be processed by DM as the other four different types of requests. As mentioned in previous chapter, the target records are processed after the source records. In order to separate the records of these two requests, DM will first send the source request and its associated address set to RECP, and hold the target request and its addresses set until receiving a message from RECP indicating that all source records have been retrieved.

The record-retrieving operation is performed by the physical-data-operation subprocess in RECP as a regular retrieve request. Instead of sending the retrieved records to the controller, control logic is used to route them to the hashing module for hashing and subsequent merging.

b. The Affected Software

Most of the operations of this phase are done by DM, CC and the Physical Data Operation of RECP in each backend. The affected software includes:

(1) We need to add control logic into DM so that the address information of the source and target request will not be sent to RECP together; and

(2) We need to add a new procedure to handle the retrieve-common request and control logic to route the results to the hashing module instead to PP.

3. The Hashing-and-storing Phase

This is the most important part of the retrieve-common request. All of the records are prepared in this phase, so they can be merged on next phase. The operations of the hashing-store phase includes:

(1) performing hashing operations on the local records,
(2) table maintenance and bucket-block tracking operations, and
(3) broadcasting (and receiving) the target records and their bucket-values to (from) the other backends.

a. The Hashing Operations

This operation is performed by the hashing procedure of the hashing module. Upon receiving the local records from the previous phase, the hashing procedure will check the record template to get the value type of the common attribute values and then apply an appropriate hashing function to hash the common attribute values. The records and their hashed bucket-values will then be passed to the bucket-block tracking procedure for further processing.

b. Table-maintenance and Bucket-block Tracking Operation

This operation is done by the bucket-block tracking procedure. A global table is maintained to store the address of all of the hashing tables for all of the different retrieve-common requests which are currently being

processed by the system. Whenever a new retrieve-common request is encountered, the bucket-block tracking procedure will create a new hashing table for that request. The logical address of the newly created hashing table is then stored into the global table. The hashing table will be deleted when the request is complete. Records are stored into buckets according to their hashed values. The information of the bucket entries and the block headers are maintained and updated by the bucket-block tracking procedure as described in the previous section.

c. Broadcasting And Receiving Target Records Between Backends

After the local target records has been hashed and processed, each backend will buffer its local target records (retrieved from the target-hashing table with their bucket values) and broadcast them to the other backends. Upon receiving those non-local target records, each backend will store them into the target-hashing table by their bucket values. A checklist is used to ensure that the target information from all of the other backends has been received.

d. The Affected Software

Since the operations of this phase are done by the hashing module; RECP is affected to the extent that this module is integrated into the RECP process. No other existing software will be affected.

4. The Merging Phase

This is the last phase of the retrieve-common operation. The local source records and the entire set of target records are compared and merged.

a. The Operation

The operations are performed by the merging procedure of the hashing module. Because the records of both tables are unsorted, they are merged by using the straightforward algorithm. The merged results are stored in a result buffer and then sent to the controller.

b. The Affected Software

Since this phase is also done by the hashing module; RECP is affected to the extent that this module is integrated into the RECP process. No other existing system software is affected.

V. THE IMPLEMENTATION

In this chapter, we describe how the retrieve-common request is integrated into the MBDS system. To successfully perform the integration, it is necessary to modify a portion of the MBDS software. Therefore, this chapter also on discussing how the MBDS software is modified for the integration and implementation of the retrieve-common operation.

In the remainder of this chapter we first describe the modified processes of the controller. Second, we describe the modified processes of each backend. Then, we present the modified MBDS message-passing facilities. Finally, we trace the execution sequence of the retrieve-common request in terms of the types of messages that are passed among the MBDS processes.

A. THE MODIFIED PROCESSES OF THE CONTROLLER

1. The Request Preparation Process (REQP)

There are two subprocesses in REQP, namely the parser and the composer. The parser parses the requests and checks for syntax errors. The composer transforms the correctly parsed requests into the form required for processing at the backends.

a. The Parser

The parser does both the lexical and the syntactical analyses of the ABDL transaction (or requests). The input to the parser is either a request or a transaction. The outputs from the parser are the error messages to the test interface, the aggregation operators to PP and the correctly parsed requests to the composer.

The lexical analysis is done by the lexical analyzer produced by IEX [Ref. 11 : p. 42]. The input to LEX is a specification of the tokens of the language(i.e., the tokens of ABDL) in the form of regular expressions and a set of subroutines which specify the actions to be taken upon recognition of the tokens. The syntactical analyzer is generated by YACC (Yet Another Compiler Complier) [Ref. 12]. The input to YACC is a specification which includes the declarations of tokens' names, the rewriting rules of the grammar, and the action program. YACC produces a C program to determine whether the input ABDL transactions (requests) are syntactically correct.

For the parser tc correctly parse the users' retrieve-common requests, we have made several modifications to the original parser subprocess. These modifications are listed below.

(1) Regular expressions for the LEX.
We have added a new set of regular expressions so that the lexical analyzer can recognize the retrieve-common request and generate appropriate tokens which in turn can be recognized and used by YACC.

(2) Grammar rules fcr YACC.
A new set of rules has been added into the original ABDL grammar sc that the parser can recognize those tokens which are generated for retrieve-common request and organize those tokens by these newly created rules.

(3) The request type.
We have added a new request type, the retrieve-common request, so that the parsed transaction can be correctly identified and properly executed by the composer and the other processes of MBDS.

(4) The action program.

The input of the retrieve-common request to the parser is in the form of a single request. The parser should be able to parse this request and generate a transaction of two retrieval requests (each of the retrieve-common request type). If the join attribute is not in the target list (of the source or the target request), the action program inserts the join attribute into the head of the target list. The extra attribute-value pairs (i.e., the join attribute-value pairs) of the retrieved records, which are going to be deleted by the merging procedure, are not to be in the results so that the merged results contains only the desired attribute-value pairs. The newly added regular expressions, grammar rules and the SSI for the modified action program are provided in Appendix A.

b. The Composer

The composer receives the correctly parsed requests from the parser and formats them into the required message format. Then, the composer broadcasts the formated messages to all of the backends for execution. We have modified the original composer program so that the composer can correctly reformat the retrieve-common request.

2. The Post Processing Process (PP)

The post processing process includes the aggregate post operation and the reply monitor. The functions of PP are described in [Ref. 11 : p. 27]. The aggregation post operation is not modified. The only modification in the reply monitor is to recognize the new request type for the retrieve-common request.

B. THE MODIFICATION OF THE BACKEND PROCESSES

As described in chapter II, one of the design issues of MBDS is to assign as much work as possible to the backends. Consequently, there are more changes in the processes of each backend than changes in the controller. The affected processes are directory management and record processing.

1. The Directory Management Process (DM)

DM receives the new transaction message for the retrieve-common request from the request composer and then performs a number of directory operations, which includes attribute search, descriptor search, cluster search, address generation and directory table maintenance. From our earlier discussion, we know that the source and target request for a retrieve-common request should not be processed concurrently by RECP. The target request must be held in DM until RECP informs DM that the source request has finished execution. Therefore, DM will first process the source request and send the request and its addresses to RECP. The target request is held in DM until RECP notifies DM that the source request is done.

At what stages of the DM processing do we hold the target request? There are several alternatives for holding the target request in DM. These alternatives are list below.

(1) Hold the target request without performing any directory operation.

(2) Hold the target request after it completes attribute search.

(3) Hold the target request after it completes attribute search and descriptor search.

(4) Hold the target request after it completes attribute search, descriptor search and cluster search.

(5) Hold the target request after it completes attribute search, descriptor search, cluster search, and address generation.

Alternatives 2, 3, 4, and 5 will generate status and directory information for the target request which must be held somewhere. Due to the large number of the possible attributes, the size of the status and directory information may be too big to be kept in the primary memory, i.e., they will have to be stored back to the secondary storage. The extra disk I/O time for moving the status and directory information in and out of the primary memory, not only slows the retrieve-common operation, but also increases the program complexity and causes many unnecessary changes to the existing software. Therefore, we choose alternative (1) to process the target request.

The algorithm for the modified DM is as follows.

Step 1: Get the next message from the message queue and find the sender of the message.

Step 2: If the sender is the controller, then go to step 5.

Step 3: If the sender is RECP, then go to step 8.

Step 4: If the sender is CC, then go to step 11.

Step 5: If this is not a retrieve-common transaction, then go to step 11.

Step 6: Identify and separate the source request and the target request from the transaction. Hold the target request and perform the directory processing on the source request.

Step 7: Send the source request with its address set to RECP. Go to step 1.

Step 8: If this is not the message which indicates the completion of retrieving all the source records, then go to step 11.

Step 9: Get the correspondent target request and perform directory processing on that target request.

Step 10: Send the target request with its address set to RECP.

Step 11: Perform the original DM operation.

The SSL for the modified DM is provided in Appendix B.

2. The Record Processing Process (RECP)

RECP receives the requests and their address sets from DM and performs the physical data operations on those requests. The original physical-data-operation subprocess includes a control function and a subfunction for each type of request. The subfunctions are invoked by the control function according to the type of request being processed.

In order to process the retrieve-common request, we have made two modifications to RECP:

(1) adding a new subfunction, the retrieve-common subfunction, into the physical-data-operation subprocess; and

(2) adding a new subprocess, the hashing module, into RECP.

a. The Retrieve-Common Subfunction

The purpose of the retrieve-common subfunction is to direct the flow of the control in the physical-data-operation subprocess so that the retrieve-common request can be processed correctly. The difference between the retrieve-common subfunction and the retrieve subfunction can be summarized as follows.

(1) The retrieve subfunction sends the retrieved records to the PP, whereas the retrieve-common subfunction sends the retrieved records to the hashing module.

(2) In addition to sending a message to CC to indicate the completion of the retrieval of physical data (as the retrieve subfunction does), the retrieve-common subfunction will send a message to notify DM that all the source records have been processed.

The algorithm for the retrieve-common subfunction is as follows.

Step 1: Reserve a result buffer.

Step 2: For each address in the set of tracks which are furnished by DM, fetch the track from the disk and place it in the track buffer in the primary memory.

Step 3: Examine the records in the buffer one-by-one. If the record is marked for deletion, disregard it. If the record does not satisfy the query, disregard it. If a record satisfies the query, then extract the values for the attribute names in the target-list of the request and store this information in the result buffer.

Step 4: When the result buffer is full, send the contents of the buffer to the hashing module.

Step 5: Repeat steps 2, 3 and 4 until there are no more addresses for the request.

Step 6: Send a message to CC to release the lock for this request. If this is a source request, then send a message to DM so that DM can process the target request.

Step 7: Free the result buffer.

The SSL for the modified control function and the retrieve-common subfunction are provided in Appendix C.

b. The Hashing Module

The hashing module performs the hashing and merge operations. The merged results are sent to the controller. The module is invoked by the retrieve-common subfunction of the physical-data-operation subprocess. There are three procedures within this module, the hashing procedure, the bucket-block tracking procedure and the merging procedure.

(1) The Hashing Procedure. The hashing procedure receives the records from the retrieve-common subfunction of the physical-data-operation subprocess and performs the hashing function on the value of the join attribute of each record. The records and their hashed results are stored in a result buffer. When the buffer is full, its contents are passed to the bucket-block tracking procedure for further processing.

The algorithm for the hashing procedure is as follows.

Step 1: Reserve a result buffer.

Step 2: Get the data type of the value of the join attribute from the record template and reserve a result buffer.

Step 3: Extract a record from the input buffer which is passed from the retrieve-common subfunction.

Step 4: Apply the appropriate hashing function to hash the value of the join attribute of the record according to data type. (See Chapter IV again.)

Step 5: Store the record and the hashed bucket value in the result buffer.

Step 6: If the result buffer is full, then send the contents of the result buffer to the bucket-block tracking procedure.

Step 7: Repeat steps 3, 4, 5 and 6 until there are no more records in the input buffer.

Step 8: Free the result buffer.

The SSL for the hashing procedure is provided in Appendix D.

(2) The Bucket-block Tracking Procedure. This procedure stores the records (both the source records and the target records) into blocks according to their bucket values and maintains one hashing table for the currently processed request and one global table to store the logical-hash-table addresses for all of the retrieve-common requests in system. The inputs to this procedure are the records and their hashed bucket values, which either come from the local hashing procedure or from the other backends. A checklist is used to ensure that the hashed results of the non-local target records are received from all of the other backends. There is also an additional disk I/O buffer used in this procedure to move the blocks of each bucket into and out of the primary memory. The outputs from this procedure are the logical addresses of the two hashing tables of the source request and the target request, which are passed to the merging procedure. The structures of the global table, hashing table, bucket, and block have been described in Chapter IV. After processing all of the local records, this procedure will group the local target records together with their bucket numbers, and then broadcast them to all of the other backends.

The algorithm for this procedure is as follows.

Step 1: Create the global table and reserve a disk I/O buffer.

Step 2: Get an input buffer of records. If the input buffer contains source records, then go to step 5.

Step 3: If the input buffer contains local target records, then go to step 6.

Step 4: If the input buffer contains the target records received from the other backends, then go tc step 8.

Step 5: Get the hashing table for the source request. Go to step 7.

Step 6: Get the hashing table for the target request.

Step 7: Store the record into a bucket and perform the bucket-block tracking operation (as described in chapter IV). Go to step 9.

Step 8: Perform the bucket-block tracking operations to insert these incoming records into the target hashing table.

Step 9: Repeat steps 2 to 8 until all records have been processed.

Step 10: If the input buffer contains local target records, then retrieve the local target records from the target hashing table bucket-by-bucket and broadcast them (with the bucket number) to the other backends.

Step 11: If the input buffer contains non-local target records, then get the logical address of the hashing table of the source request. Pass the logical address of the hashing tables of the source request and the target request to the merging prccedure for the merging operation.

The SSL for this procedure is provided in Appendix E.

(3) The Merging Procedure. This procedure does three functions:

(1) fetching the hashing tables of the source request and the target request by their logical addresses which have been prcvided by the bucket-block tracking procedure;

(2) performing the merging operation on the records of both hashing tables (as described in chapter IV); and

(3) sending the merged results to the controller.

The merged results contains only the attribute-value pairs whose attribute names are specified in the target-lists (either the source request or the target request). The extra attribute-value pairs (i.e., the join attributes and their vales, which have been added into the target lists by the parser) are deleted by this procedure. The SSL for the merging procedure is provided in Appendix E.

C. THE MODIFIED MESSAGE-PASSING FACILITIES

In Chapter II we have introduced the general format and the different types of MBDS messages (see Figure 2.3 and Figure 2.4). In order to accomplish the retrieve-common request we have added two new message types which are shown in Figure 5.1.

D. EXECUTION OF A RETRIEVE-COMMON REQUEST--VIEWED VIA MESSAGE-PASSING

In this section we describe the sequence of actions for executing the retrieve-common request as it moves through MBDS. The sequence of actions are described in terms of the types of messages passed between the MBDS processes: REQP, PP, DM, RECP and CC. The order in which message are passed is denoted alphabetically ('a' is first). The digit following the ordering letter will be the message type as shown in Figures 2.4 and 5.1.

The sequence of actions for a retrieve-common request is shown in Figure 5.2. First the retrieve-common request comes to REQP from the host (a1). REQP sends two messages to PP: the number of requests in the transaction (b3) and the aggregate operator of the request (c4). The third message

Message Type	: (32) Hashed Target Records
Source	: Record Processing
Destination	: Record Processing (other backends)
Explanation	: This message contains the bucket numbers of the target hashing table and all of the target records associated with their buckets.
Message Type	: (33) Source Retrieve Finished
Source	: Record Processing
Destination	: Directory Management (same backend)
Explanation	: This message is used to notify Directory Management that all of the source records have been retrieved. DM can then begin processing the target request.

Figure 5.1 The New MBDS Message-Types.

sent by REQP is the parsed traffic unit which goes to DM in the backends (d6). DM sends the type-C attributes needed by the retrieve-common request to CC (e20). Once an attribute is locked and descriptor search can be performed, CC signals DM (f26). DM then process the source request (target request is now held). DM performs descriptor search and signals CC to release the lock on that attribute (g23). DM sends the descriptor ids for the request to the other backends (h15). The DM processes in the other backends send their descriptor ids to the DM process residing in this backend (i15). DM then uses its own descriptors and the descriptors received from the other backends to form descriptor-id groups. DM now sends the descriptor-id groups for the source request to

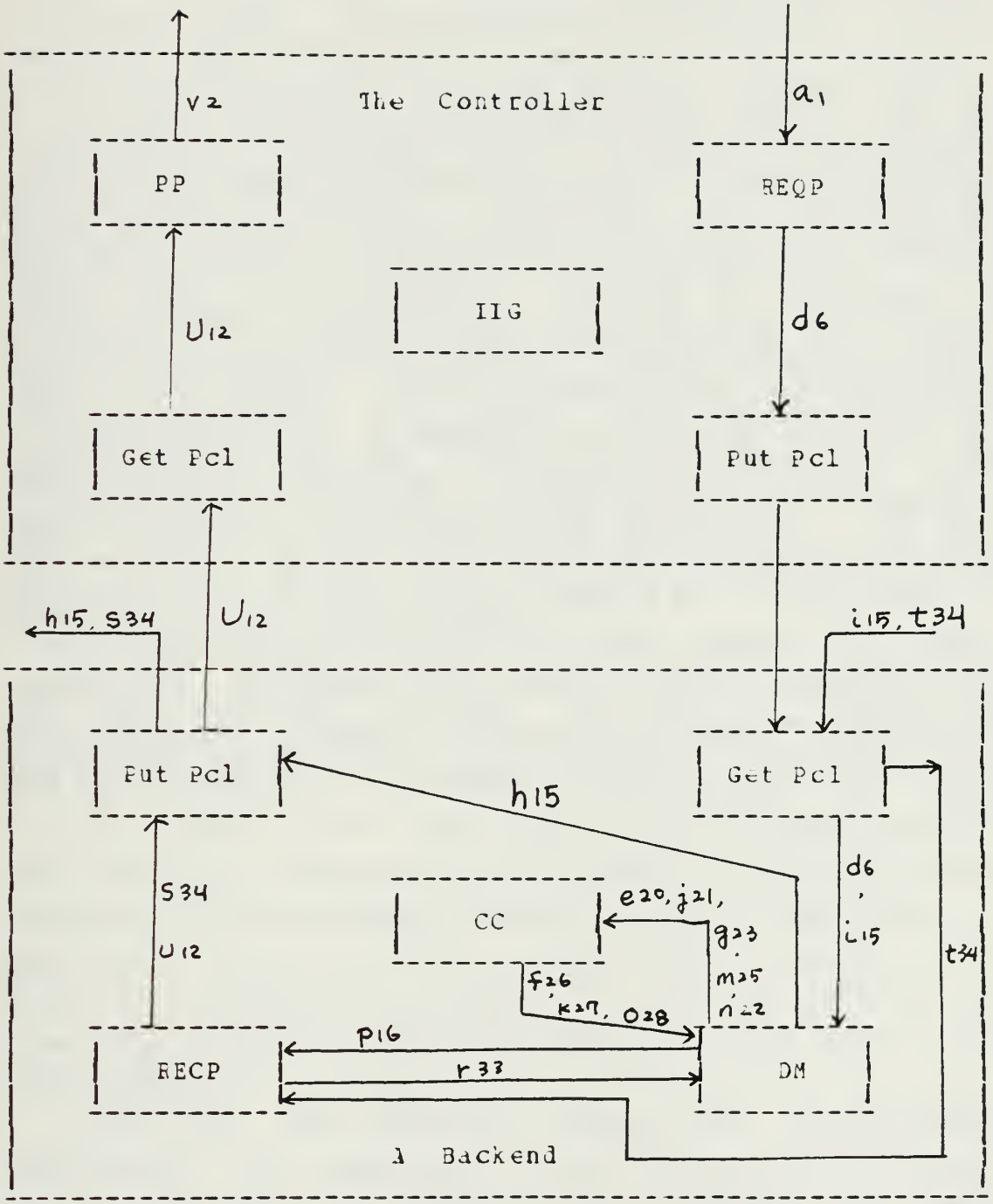


Figure 5.2 The Sequence of Messages for Executing a Retrieve-common Request.

CC (j21). Once the descriptor-id groups are locked and cluster search can be performed, CC signals DM (k27). DM then performs cluster search and signals CC to release the locks on the descriptor-id groups (m25). Next, DM sends the cluster ids for the retrieval to CC (n22). Once the cluster ids are locked, and the request can proceed with address generation and the rest of the source-request execution, CC signals DM (o28). DM then performs address generation and sends the source request and the address set to RECP (p16). Once the retrieval request has executed properly, RECP sends a message to DM to start processing the target request (r33). DM processes the target request in the same way of processing the source request (i.e., phases e20 to p16). The retrieved records are processed by the hashing module in RECP. Once the local target records have been processed properly, the hashing module broadcasts the hashed target records (grouped by bucket numbers) to the other backends via RECP (s34). The hashing modules in the other backends sends their hashed target records to the hashing module of this backend (t34). Once the comparing and merging operations performed by the hashing module, the results are sent to PP (u2). PP then forwards the results to the host (v2).

VI. CONCLUSION

A. REVIEW AND SUMMARY

The multi-backend database system (MBDS) in the Laboratory for Database System Research at the Naval Postgraduate School is designed to overcome the performance-gain and capacity-growth problems of either the traditional database system or the single-backend-software-database system. The original MBDS supported four primary operations, namely, RETRIEVE, DELETE, UPDATE and INSERT. This thesis presented the design and implementation of the fifth primary operation, the RETRIEVE-COMMON operation. The retrieve-common operation is used to merge two files by common attributes. Our major goal is to maximize the utilization and minimize the affects to the existing system.

We have analyzed several possible design alternatives and then selected the best one for our design and implementation approach. The key issues for the selections are the cohesion to the design requirements, the design issues of MBDS and the time complexities of implementation. Our design and implementation is based on the bucket-hashing approach. Each backend performs partial merge with its portion of source records and the entire set of target records, sending its results to the controller. The controller forwards the final results to the user at the host computer.

Based on the selected design and implementation approaches, the operations of the retrieve-common request are executed in four phases, the request-preprocessing phase, the record-retrieving phase, the hashing-and-storing

phase and the merging phase. The retrieve-common requests is first parsed to be a transaction of two retrieval requests (each of the retrieve-common type request) by the parser. Then, the parsed requests are reformated into required message formats and broadcasted to all the backends by the composer of the controller. Each backend receives the formated messages of the transaction, separates the source request and the target request and then performs the directory operations and retrieves the records according to the queries specified in the requests. The retrieved records of the source record set and the records of the target record set are separately hashed on their common attribute values and then stored into buckets of the source hashing table and the target hashing table, respectively. The hashed records of the source buckets and the records of the target buckets are compared and merged bucket-by-bucket. The merged results are sent to the controller from all of the backends. The controller then forwards the results to the host computer. In order to accomplish the operations of the retrieve-common request, we have designed a hashing module into the record-processing process of each backend.

For integrating our design into MBDS, we have made several modifications. These are:

(1) the message-passing facilities,
(2) the parser of the request-preparation process of the controller, and
(3) the directory-management process and the record-processing process of each backend.

The algorithms for the modifications and the program specifications (SSL) are also provided in Character IV, V and Appendices.

B. FUTURE WORK

The next step in the design and implementation of the retrieve-common operation is the modification of the MBDS software according to the SSL given in the appendices. There are two classes of modifications. First, existing software is updated to reflect the changes necessary for the retrieve-common operation. In the system, new message types must be defined, the request-prepatation and post-processing processes of the controller are changed, and the directory-management process is changed to correctly sequence and execute the retrieve-common request. Second, new software is written to handle the processing of the retrieve-common request, i.e., the hashing module. In the system, the software for the hashing module is coded tested, and integrated into the record-processing process of each backend.

APPENDIX A

THE MODIFIED REQUEST PREPARATION PROGRAM SPECIFICATIONS

In this appendix, we present only the modified portions of the Request Preparation process. The original SSI is in [Ref. 11 : p.87].

A. THE LEX MODIFICATIONS

```
/*********************************************************
*                                                        *
*  We have added the regular expression for the token    *
*                                                        *
*  COMMON into LEX.  The rest of LEX remains unchanged.  *
*                                                        *
*  The original specification is in the lsrc file.       *
*                                                        *
*********************************************************/
     .
     .  (The original lscr specifications.)
     .
BY        {
           return(TOKBY);
          }
COMMON    {
           return(TOKCOM);
          }
"<="      {
           return(LE);
          }
     .
     .   (The original lscr specifications.)
     .
```

B. THE YACC MODIFICATIONS

In this section, we present only the SSL for the modified portion of the parser. The original program is in the ysource file.

```
procedure yyparse();
    /*******************************************************
    *  This procedure is used to parse the output of LEX.  *
    *  The modification of the yyparse procedure converts  *
    *  the retrieve-common request from a single request   *
    *  into a transaction of two requests.                 *
    *                                                      *
    *  Data structures and variables used in this          *
    *  procedure:                                          *
    *   1. No new data structures are introduced by this   *
    *      modification.                                   *
    *   2. com_flag_1, com_flag_2, com_flag_3, com_flag:   *
    *      Boolean variables which are used indicate the   *
    *      different conditions of the retrieve_common     *
    *      request.                                        *
    *   3. new_tbl_ptr:                                    *
    *      A pointer to a request table.                   *
    *      The request table is defined in the commdata.def*
    *      file as a REQtbl_definition structure.          *
    *   4. com_atrb_1, com_atrb_2:                         *
    *      Character strings to hold the common attribute. *
    *******************************************************/

    /* The following is the modified portion of yysource.*/

    /* Add a new token in the specification. */
    %token [str] TOKCCM   /* common  */

    /* Add new derivations and program specifications.  */

    transaction : beg_tran lines
                  /* No changes in this part  */
```

```
                /* of the transaction rule. */
              | beg_single_req line
                if com_flag
                   then
                     /* This is a retrieve-common
                        request. */
                        Perform the operations which are
                        specified under the beg_tran
                        lines;
                   else
                     /* Perform original operations. */
                end if;

end_req     : EOR
              /* Clear the com_flags. */
              com_flag = false;
              com_flag_3 = false;

 req_forms  : delete query
            |     ...
            |     .../* These are the
                         original derivations. */
            |     ...
            |  req_forms common target_list req_forms;

 common     : TOKCOM
                perform CHECK_REQUEST_TYPE(req_tbl,OK);
                        /* Check if the first request is
                           a  retrieve. */
                if OK
                   then
                     com_flag = com_flag_1 = true;
                   else
                     perform ERROR_PROCEDURE;
                end if;
 attribute  : LETTERFIRST
```

```
if com_flag_1
   then
     /* This attribute is the common
        attribute of the source
        request. Copy the attribute
        into com_atrb_1. */
     perform strcpy(com_atrb_1,
                    attribute);
     /* Put the common attribute of
        the source request into
        the target list and
        convert the request table from
        the form of single request to
        the form of a transaction. */
     perform CONVERT(tbl_ptr->req_tbl,
                     com_atrb_1,
                  traf_id, req_cnt,
               new_tbl_ptr->req_tbl);
     com_flag_2 = true;
     com_flag_1 = false;
     /* com_flag = true */
   else
     if com_flag_2
        then
          /* This attribute is the
             common attribute of the
             target request. */
          com_atrb_2 = strcpy(attribute);
          com_flag_3 = true;
          com_flag_2 = false;
        else
          if com_flag_3 = true;
             then
               /* This is the first
                  attribute of the target
```

```
                                list of the target
                                request. */
                                insert com_atrb_2 intc the
                                target request table;
                                insert the attribute into
                                the target request table;
                            end if;
                            /* Perform the original
                               operations. */
                      end if;
              end;

retrieve  : TOKRETRIEVE
            if ccm_flag_3
               then
                 perform ERROR_PROCEDURE;
               else
                 if com_flag
                    then
                      /* Change the type to be
                         RETRIEVE_COMMON.  */
                 end if;
             end if;
            /* Perform the original operations. */

delete    : TOKDELETE
            if com_flag
               then
                 perform ERROR_PROCEDURE();
               else
                 /* Perform the original operations. */
            end if;

insert    : TOKINSERT
            if ccm_flag
               then
```

```
                    perform ERROR_PROCEDURE();
                  else
                    /* Perform the original operations. */
                end if;

    update    : TOKUPDATE
                if ccm_flag
                  then
                    perform ERROR_PROCEDURE():
                  else
                    /* Perform the original operations. */
                end if;

    /* Perform the original operations. */
end procedure yyparse;

procedure CONVERT(input: source_req_table, source_com_atr,
                         traf_id, request_number,
                         index_req_ptr;
                  output: target_req_table, request_number,
                          index_req_ptr);
    /*****************************************************
    *  This procedure is used to rearrange the contents    *
    *  of the request table of a request which is the      *
    *  source retrieve of a RETRIEVE_COMMON request.       *
    *  This procedure performs the following tasks:        *
    *    1. Rearrange the source request table.            *
    *    2. Make the common attribute of the source request*
    *       the first attribute of the target list.        *
    *    3. Create a request table for the target request  *
    *       and return it to the calling procedure.        *
    *                                                      *
    *  Data structures and variables used in this          *
    *  procedure are:                                      *
    *    1. source_req_table, target_req_table:           *
    *       The request tables of the source request and   *
```

```
*       the target request.                                    *
*    2. new_table:                                             *
*       An array of Reqtbl_definition structures.              *
*    3. traf_id:                                               *
*       A  character string which is the traffic id of         *
*       a transaction.                                         *
*    4. request_number:                                        *
*       An integer which is used to indicate the               *
*       number of requests in a traffic unit.                  *
*    5. index_req_ptr:                                         *
*       A pointer to a parsed traffic unit, which  is          *
*       an  array of Reqtbl_definition structures.             *
*    6. source_ccm_atr:                                        *
*       A character string which is the common                 *
*       attribute of the source request.                       *
*****************************************************/

/* Use a new request table, new_table to hold the
   contents of the source_req_table, */
 new_table[0] = BCR;
 new_table[1] = str_to_num(traf_id);
 new_table[2] = request_number;
 new_table[3] = rcuttype; /* Defined in yyparse().*/
 new_table[4] = RETRIEVE_COMMON;
 /* Copy the contents of the source request table into
    the new_table. */
 i = 5;
 repeat
   new_table[i] = source_req_table[i];
   i = i+1;
 until source_req_table[i] = EOQ;
 /* Insert the common attribute into the new_table.*/
 new_table[i] = scurce_com_atr;
 i = i+1;
 /* Ccpy the rest of the source_req_table into
```

```
        the new_table. */
      repeat
        new_table[i] = source_req_table[i-1];
        i = i+1;
      until source_req_table[i-1] = null;
      /* Put an end-of-request marker, EOR,
         into the new_table. */
      new_table[i] = EOR;
      /* Copy the new_table into the source_req_table. */
      i = 0;
      repeat
        source_req_table[i] = new_table[i];
        i = i+1;
      until source_req_table[i] = EOR;

    /* Increase the request number, and create a request
       table for the target request. */
    request_number = request_number+1;
    perform ALLOCATE_REQ_TABLE(target_req_table);
    /* Put the target_req_table into the
       parsed traffic unit. */
    index_req_ptr->req_tbl[request_number-1]
                          = target_req_table;
    /* Return the request number,  target_req_table and
       index_req_ptr to the calling procedure. */
end procedure CONVERT;
```

```
procedure CHECK_REQUEST_TYPE(input: req_tbl; output: ok);
    /*****************************************************
    * This procedure is used to check the syntax of a    *
    * retrieve_common request.  If the request type is   *
    * not retrieve, set OK to false. Otherwise, set OK   *
    * to true.     Return OK to the calling procedure.   *
    *****************************************************/
end procedure CHECK_REQUEST_TYPE;



procedure ERROR_PROCEDURE();
    /*****************************************************
    *  This procedure is used whenever there is a syntax  *
    *  error in the request.                              *
    *  This procedure will print an error message and     *
    *  terminate the parser operations.                   *
    *****************************************************
end procedure ERROR_PROCEDURE;
```

APPENDIX B
THE MODIFIED DIRECTORY MANAGEMENT PROGRAM SPECIFICATIONS

The original SSL for the Directory Management process is in [Ref. 13 : p. 82-102]. In this appendix, we present only those procedures which are affected by the retrieve-ccmmon request.

```
procedure DM_ParesedTrafUnit();
    /*******************************************************
    *  This procedure is used when Request Preparation    *
    *  (REQP) sends a traffic unit to Directory           *
    *  Management (DM). The original procedure is in      *
    *  the tu.c file.                                     *
    *  We add an if statement to differentiate between    *
    *  the retrieve-ccmmon request type and the other     *
    *  request types.                                     *
    *  No new variables are introduced in this procedure. *
    *******************************************************/

    /* Get a pointer to the parsed traffic unit. */
    ti_ptr = DM_R$ParsedTrafUnit();
    /* Get a pointer to the record template
       of this traffic unit. */
    tmpl_ptr = get_tmpl_ptr(ti_ptr->ti_dbid);
    /* Get a pointer to the attribute table. */
    AT = AT_lookuptbl(ti_ptr->ti_dbid);
    /* Get the type-c attributes for the traffic unit
       and send them to DS_CC. */
    perform DM_TypeC_Attrs_TrafUnit();
    /* Process the requests of this traffic unit. */
```

```
    ri_ptr = ti_ptr -> ti_first_req_pointer;
    /* Get the type of the first request of
       this traffic unit.*/
    if req_type = RETRIEVE_COMMON
       then
         /* We will only process the source request. */
         /* The target request will not be processed */
         /* until the record-processing process has  */
         /* retrieved all of the source records.     */
         /* Perform the descriptor search processing. */
         done = NINS_SR_DESC(&rie, ri_ptr, tmpl_ptr, AT);
         if done
            then
              /* Broadcast the descriptor ids to the
                 other backends. */
              DM_Broadcast_DIDs(&rid);
         end if;
       else
         /* This is not a retrieve-common transaction, so
            process the requests of the traffic unit
            one-by-one. */
    end if;
end procedure DM_ParesedTrafUnit;


procedure DM_RecP_Msg()
    /*****************************************************
     *  This procedure is used when there is a message   *
     *  for DM from RECP (in the same backend).          *
     *                                                   *
     *  We add a new message type to indicate  that all  *
     *  of the source records have been retrieved.       *
     *                                                   *
     *  No new data structures or variables are used.    *
     *  The original procedure is called by              *
```

```
     *  DM_THIS_BE_MSG() and is in the dirman.c file.     *
     ****************************************************/

    /* Get the message type. */
    MsgType = DM_R$Type;
    switch (MsgType)
      case OldNewValue:
           perform DM_OldNewValues();
      case UpdFinished:
           perform DM_UpdFinished();
      case Source_finished:
           /* This is the message which indicates the
              completion of the retrieval of all the
              source records. */
           perform DM_Source_finished(msg);
    end switch;
end procedure DM_RecP_Msg;


procedure DM_Source_finished(input: message);
    /*****************************************************
    *  This procedure is used when DM receives a messages, *
    *  from  RECP, which indicates the completion of the   *
    *  retrieval of all of the source records. DM is now   *
    *  ready to process the target request.                *
    *                                                      *
    *  This procedure is called by DM_Recp_msg().          *
    *****************************************************/

    /* Receive the request id from the message. */
    perform DM_R$Rid(source_req_id);
    /* Get a pointer to the traf_info entry by the
       source_req_id.*/
    ti_ptr = DM_TiFind(source_req_id);

    /* Get a pointer to the req_info entry for the source
       request. */
```

```
    source_req_info_ptr = DM_RiFind(req_id, ti_ptr);

    /* Get a pointer to the req_info entry for the target
       request by the source_req_info_ptr. */
    target_ri_ptr = scurce_req_info_ptr->next_req_info;

    /* Get the request id of the target request. */
    target_req_id = Find_request_id(target_ri_ptr);

    /* Perform the directory operations on the
       target request.*/
    /* Get the record template for the target request.*/
    tmpl_ptr = get_tmpl_ptr(ti_ptr->ti_tbid);
    /* Get a pointer to the attribute table. */
    AT = AT_lookuptbl(ti_ptr->ti_dbid);
    /* Perform the descriptor search processing. */
    dcne = NINS_SR_DESC(&rid, ri_ptr, tmpt_ptr, AT);
    if dcne
       then
         /* Broadcast the descriptor ids to the other
            backends. */
         perform DM_Broadcast_DIDs(&rid);
    end;
end procedure DM_Source_finished;
```

APPENDIX C

THE MODIFIED RECORD PROCESSING PROGRAM SPECIFICATIONS

In this part of the appendix, we have added the retrieve-common subfunction into the control function of the physical-data-operation subprocess of the record-processing process (RECP). We have presented only the modified portion of the original RECP in this appendix.

```
procedure ReqProcessing(input: MsgType) ;
   /*****************************************************
   *                                                    *
   * This procedure is used to process requests according *
   *                                                    *
   * to the request type.                               *
   *                                                    *
   * We add the retrieve-common request type into the   *
   *                                                    *
   * switch statements as one of the optional cases.    *
   *                                                    *
   * This procedure is called by the procedure RP_DM. The *
   *                                                    *
   * original procedure is in the reproc.c file.        *
   *                                                    *
   *****************************************************/

    /* Get the request type. */
    switch (request_type)
         RETRIEVE_COMMON:
              perform ST_RetDel();
              /* From this point, we ues the same
                 procedures as used for the
                 RETRIEVE request processing. */
    /* Now, back to the original ReqProcessing(). */
end procedure ReqProcessing;
```

```
procedure RP_ReadCompleted();
    /*****************************************************
     *                                                   *
     * This procedure is used when a physical read is    *
     *                                                   *
     * ccmpleted.  We add the retrieve-common request    *
     *                                                   *
     * type into its switch statements as one of  the    *
     *                                                   *
     * the request types cases.                          *
     *                                                   *
     * This procedure is called by the procedure RP_RP.  *
     *                                                   *
     * The original procedure is in the recproc.c file.  *
     *                                                   *
     *****************************************************/
    /* Get the request type of this request. */
    switch (request_type)
        RETRIEVE_COMMON :
            perform RC_Ret();
        RETRIEVE:
            perform RC_Ret();
            /* Now, back to the original processing. */
    end switch;
end procedure RP_ReadCompleted;


procedure RB$SEND_COMPLETION(input: RB_ptr, reqtype);
    /*****************************************************
     *  This procedure does the following tasks:         *
     *    1. Send the contents of the result buffer to   *
     *       either the hashing module or the controller,*
     *       depending on the request type.              *
     *    2. If this is a source request of a retrieve-  *
     *       common request, then send a message to DM   *
     *       indicating that all of the source records   *
     *       have been retrieved.                        *
     *    3. Send a message to CC to release the locks on *
     *       the database for this request.              *
     *    4. Free the result buffer space after the      *
     *       contents of the result buffer have been sent.*
     *                                                   *
```

```
 *  All of the data structures ans variables are the  *
 *  same as the original procedure.                   *
 *  This procedure is called by the procedure         *
 *  RC_Ret().                                         *
 *  The original procedure is in the recproc.c file.  *
 ****************************************************/

/* Get the request id by the result buffer pointer
   RB_ptr.*/
request_id = RB_ptr->RB_rid;
if reqtype = RETRIEVE_COMMCN
   then
     if the result_buffer is full
        then
          /* Send the contents of the result buffer */
          /* to the hashing module and reinitialize */
          /* the buffer size to 0.                  */
          HASH_FUNC(request_id, result, result_length);
          result_length = 0;
      end if;
      if this is the last result buffer
         for this request
         then
           /* Send the result buffer to the
              hashing module. */
           perform HASH_FUNC(request_id, result,
                             result_length);
           if this is a source request
              then
                /* Send a message to DM indicating  */
                /* that all of the source records   */
                /* have been retrieved.             */
                perform DM_FinReq$RP_S(request_id);
           end if;
           /* Free the result buffer space. */
```

```
                perform Recp_free(request_id);
                /* Send a message to CC to    */
                /* release the locks for this    */
                /* request. */
                perform CC_FinReq$RP_S(request_id);
            end if;
          else
            /* This request is not a retrieve-common
               request.
               Now, back to the original processing. */
    end if;
end procedure RB$SEND_COMPLETION;


procedure XTRACT(input: TRACK_BUFFER, indexB, result2,
                        request, tmpl_ptr, target_ptr;
                  output: result2);
    /*****************************************************
     *  This procedure extracts the attribute names and   *
     *  values  which  correspondend  tc the target list  *
     *  cf a record.                                      *
     *  This procedure is called by the procedure         *
     *  $RETR_PROCESSING().                               *
     *  The original procedure is in the rbabs.c file.    *
     *  We add an end-of-record marker, EOR, at the end   *
     *  of every reccrd.                                  *
     *****************************************************/

    /* Process all   statements of the original procedure
       until the end of the outermost while loop. */
    /* Add the following processing. */
    if the reqtype = RETRIEVE_COMMON
       then
         put the EORecord marker into the result buffer;
    end if;
    /* Now, back to the original processing. */
```

```
end procedure XTRACT;


procedure RB$PUT_SEND(input: RESULT_BUFFER, result,
                          length_of_result);
    /*****************************************************
     *  This procedure puts the results for a request     *
     *  into  the result buffer. If the result buffer is  *
     *  full, then the contents of the buffer are sent to *
     *  the controller or the hashing module and the      *
     *  length of the buffer is set to 0.                 *
     *  This procedure is called by the procedure         *
     *  RETR_PROCESSING().                                *
     *  The original procedure is in the rbabs.c file.    *
     *****************************************************/


    if the result buffer is full
       then
         /* Find the request type in the result buffer.*/
         reqtype = FIND_req_type(result_buffer);
         if reqtype = RETRIEVE_COMMON
            then
              /* Send the results to hashing module. */
              perform HASH_FUNC(result_buffer);
            else
              /* Send the results to the controller. */
              perform RES$CNTL$RP_S(request_id,results,
                                    length_of_result);
         end if;
         length_of_result = 0;
       else
         /* Store the results into the result buffer. */
         /* Now, back to the original processing.    */
    end if;
end procedure RB$PUT_SEND;
```

```
procedure RP_CNL_ANOTHER_BE_MSG();
    /*******************************************************
     *  The purpose of this procedure is to process       *
     *  the messages received from the controller or      *
     *  the other backends.                               *
     *  This procedure is modified for processing the      *
     *  the hashed information of the non-local target    *
     *  records.                                          *
     *  The original procedure is in the reproc.c file.   *
     *******************************************************/

    /* Get the  message type. */
    perform MsgType = Type$RP_R;
    case MsgType of
         Bucket_info:
              /* This message is the hashed information  */
              /* for the non-local target records. */

              perform PROCESS_BE_TARGET();
              /* This procedure should return the sender,*/
              /* the request_id of the target request    */
              /* and whether or not this is the last     */
              /* message from this backend.              */

              /* Check to see if all the target records  */
              /* of all the other backends have been     */
              /* received.                               */
              if LAST_MSG
                 then
                   perform CHECK_RECEIVE_MSG(sender,
                                request_id, ALL_RECEIVED);
              end if;
              if ALL_RECEIVED
                 then
```

```
                    perform START_TO_MERGE(request_id);
                    /* The called routine will perform   */
                    /* the merging operation and send the */
                    /* results to the controller.        */
                end if;

    /*      Now, back to the original processing.        */
    end case;
end procedure RP_CNL_ANOTHER_BE_MSG;

procedure PROCESS_BE_TARGET(input: message;
                            output: sender, request_id
                                    LAST_RECORD);
    /*****************************************************
     *  This procedure is called to process the message  *
     *  which contains the hashed bucket information of  *
     *  the non-local target records.                    *
     *  This procedure will return the sender of the     *
     *  message, the request id of those non-local       *
     *  records and a boolean variable, LAST_RECORD, to  *
     *  indicate that all of the target records from the *
     *  sending backend have been received.              *
     *                                                   *
     *  Data structures and variables used in this       *
     *  procedure are:                                   *
     *    1. LAST_RECCRD: A boolean variable which is    *
     *                    used to indicate the end of    *
     *                    this request.                  *
     *    2. message: A character string which is used   *
     *                to store the hashed results of     *
     *                target records and is sent from    *
     *                the other backends.                *
     *****************************************************/

    /* Get the sender of the message. */
    perform GET_MSG_SENDER(sender);
    /* Get the request id of the request. */
```

```
    perform GET_REQUEST_ID(request_id);
    /* Now, check the global table to find the address */
    /* of the hashing table for this request. */
    perform CHECK_GLOBAL_TABLE(request_id, hash_table,
                               NEW_REQUEST);
    NEW_RECORD = true;
    /* Since the message is an array of characters,     */
    /* we have to bypass the header to get the record   */
    /* information. If this message is the last message */
    /* of the sending backend, then there will be an    */
    /* end-of-request marker, EORequest, in the front   */
    /* of the end-of-message marker.                    */

    I = the_integer_which_stands_for
        _the_index_where_record_start;
    /* Gets the bucket_numbers and their associated   */
    /* records from the message, then insert them into  */
    /* correct buckets of the hashing table.          */

    while ((not end of message) or (not end of request)) do
      perform GET_BUCKET_NUMBER(message, I, bucket_value);
      /* Get the bucket number of the record and the   */
      /* record itself from the message, and then      */
      /* store the record into the appropriate bucket  */
      /* of the hashing table by using the */
      /* bucket number. */
      perform GET_A_RECORD_SET(message,I,set);
      perform STORE_RECORD_IN_HASH_TABLE(hash_table,
                          bucket_number, set, NEW_RECORD);
      NEW_RECORD = false;
    end while;
    if EORequest
       then LAST_RECORD = true;
       else LAST_RECORD = false;
    end if;
end procedure PROCESS_BE_TARGET;
```

```
procedure START_TO_MERGE(input: request_id);
    /*****************************************************
     * This procedure is called when the target record   *
     * set has been received from all  of the other      *
     * backends.                                         *
     * The input request_id  is the request id of the    *
     * target request.                                   *
     * The data structures and the variables used in     *
     * this procedure are:                               *
     *   1. TARGET_TABLE : The hashing table for the     *
     *                     target request.               *
     *   2. SOURCE_TABLE : The hashing table for the     *
     *                     source request.               *
     *   3. target_id: The request id of the target      *
     *                 request.                          *
     *   4. source_id: The request id of the source      *
     *                 request.                          *
     *****************************************************/

    target_id = request_id;
    /* Get the source request id.                        */
    perform GET_SOURCE_ID(target_id, source_id);
    /* Get the hashing table of the source request.       */
    perform CHECK_GLOBAL_TABLE(source_id, global_table
                               source_hash_table,
                               NEW_REQUEST);
    /* Get the hashing table of the target request.      */
    perform CHECK_GLOBAL_TABLE(target_id, global_table
                               target_hash_table,
                               NEW_REQUEST);
    /* Merge the records of these two requests and send */
    /* the results to the controller.                    */
    perform MERGE(source_id, source_hash_table.address
```

```
                    target_hash_table.address);
end procedure START_IC_MERGE;


procedure GET_SOURCE_ID(input: request_id;
                        output:request_id);
    /*****************************************************
     * This procedure is used to find the request id for *
     * the source request by using the request id of the *
     * target request.                                   *
     * Recall that the source request and the target     *
     * request has the same traffic id, the difference   *
     * between them is that the request number of the    *
     * source request is less than that of target        *
     * request by 1.                                     *
     *****************************************************/
end procedure GET_SOURCE_ID;
```

```
procedure CHECK_RECEIVE_MSG(input: sender, request_id;
                            output: ALL_RECEIVED);
    /*************************************************
     * This procedure is used to check whether all    *
     * of  the non-local target records have been      *
     * retrieved from all of the other backends for    *
     * a particular request.  If all of the non-local  *
     * target records have been received, then         *
     * AIL_RECEIVED is set to true. Otherwise,         *
     * AIL_RECEIVED is set to false.                   *
     *************************************************/
end procedure CHECK_RECEIVE_MSG;



procedure CHECK_GLOBAI_TABLE(input:request_id;
                             output: hash_table,
                                     NEW_REQUEST);
    /*************************************************
     * This procedure is used to check whether a request *
     * is a new request by checking if the request id is *
     * in the global table. If the id is found, then set *
     * the value of NEW_REQUEST to false and return the  *
     * NEW_VALUE and the hash_table of of the request.   *
     * This procedure has been defined in HASH_FUNC().   *
     *************************************************/
end procedure CHECK_GIOBAL_TABLE;
```

```
procedure GET_BUCKET_NUMBER(input: message, index;
                            output: index, bucket_number);
     /*************************************************
     *  This procedure is used to extract the bucket    *
     *  numbers from the message, then return the       *
     *  bucket_number and the incremented index to its  *
     *  caller.                                         *
     *  Data structures and variables used in this      *
     *  procedure:                                      *
     *    1. bucket: A character string representation  *
     *               of the bucket number.              *
     *    2. j: A general purpose index.                *
     *************************************************/
    j = 0;
    repeat
      bucket[j] = message[index];
      index = index+1;
      j = j+1;
    until message[i] = EOV;
    perform STRING_TO_INTEGER(bucket, bucket_number);
end procedure GET_BUCKET_NUMBER;
```

```
procedure GET_A_RECORD_SET(input: message, I;
                           output: set);
    /*****************************************************
     * This procedure is used to extract the common      *
     * attribute value of a record and the record itself*
     * from the message which contains the hashed bucket*
     * information of the non-local target records.      *
     *                                                   *
     * The data structures and the variables used in     *
     * this procedure are:                               *
     *   1. set: A array which contains the common       *
     *           attribute value of a record and the     *
     *           record itself.                          *
     *   2. j: A general purpose index.                  *
     *****************************************************/
    J = 0;
    repeat
      set[J] = message[I];
      I = I+1;
      J = J+1;
    until message[I-1] = EORecord;
end procedure GET_A_RECORD_SET;
```

APPENDIX D

THE HASHING PROCEDURE PROGRAM SPECIFICATIONS

```
Procedure HASH_FUNCTICN(input: request_id, result, length;
                        output: request_id, hashed_result,
                                length_hashed_result);
   /*****************************************************
    * The purpose of this procedure is to hash the value *
    * of the join   attribute into a bucket of the hash  *
    * table.                                             *
    * A hash buffer is reserved to store the hashed      *
    * results.                                           *
    * Data structures and variables used in this         *
    * procedure are:                                     *
    *   1. hash_buffer: A variable of the data type      *
    *                   hashing_buffer  which is used    *
    *                   to stored the records and their  *
    *                   hashed bucket values, and is     *
    *                   defined in hashing_module.def.   *
    *   2. RP_rid_info: The information for a request.   *
    *                   This structure is defined in     *
    *                   the commdata.def file.           *
    *   3. RP_rid_ptr:   A pointer to the data structure *
    *                    of type RP_rid_info.            *
    *   4. req_tbl_ptr: A pointer to a request table.    *
    *                   The request table is defined in  *
    *                   the commdata.def file as a       *
    *                   REQtbl_definition structure.     *
    *   5. temp_entry: A variable of data type rt_ntry   *
    *                  which is defined in commdata.def. *
    *   6. tem_ptr: A pointer to temp_entry.             *
    *   7. rt_enrty: A pointer to a field of RP_rid_info.*
    *                The type of this field is rt_ntry.  *
```

```
 ****************************************************/

/* Check if the request id  is a new request.        */
if new request
   then
     /* Get the record template to find the value    */
     /* type (i.e., integer, string or float) of the */
     /* common attribute value.                      */
     perform FIND_RP_rid_info(request_id,RP_rid_ptr);
     /* Get a pointer to the request table from the  */
     /* RP_rid_info.  */
     req_tbl_ptr = RP_rid_ptr -> RP_ri_req;
     /* Find the attribute name from
        the request table. */
     perform FIND_COMMON_ATTRIBUTE(req_tbl_ptr,
                                   attribute_name);
     /* Get a pointer to  the entry   */
     /* of the template for the common attribute.    */
     tem_ptr = RP_rid_ptr -> RP_ri_tmpl_ptr -> rt_entry;
     /* Get the value type of the common attribute    */
     /* from the record template.                     */
     if tem_ptr->temp_entry.value_data_type = 's'
        then
          value_type = string;
        else
          /* If the  value type is integer, then     */
          /* we decide which hashing function to     */
          /* use.                                    */
          MAX = tem_ptr.value_c1; /* The possible    */
                                  /* maximum value   */
                                  /* for this        */
                                  /* attribute.      */
          MIN = tem_ptr.value_c2; /* The possible    */
                                  /* minimum value    */
                                  /* for this         */
```

```
                                        /* attribute.        */
            if (MAX-MIN) < the_number_of_buckets
               then
                value_type = small_integer
               else
                range = (MAX-MIN) / the_number_of_buckets;
                value_type = large_integer;
            end if;
        end if;
    end if;
    /* Allocate a buffer to store the hashed results. */
    perform ALLOCATE_HASH_BUFFER(Hash_buffer);
           /*  Note: we may not want to call this */
           /*        routine at this point.       */
    switch (value_type)
      case string:
        perform STRING_HASH(result,
                            hash_buffer);
      case small_integer:
        perform SMALL_INTEGER_HASH(result, MIN
                                   hash_buffer);
      case large_integer:
        perform LARGE_INTEGER_HASH(result, MIN,
                                   range,
                                   hash_buffer);
    end switch;
end procedure HASH_FUNC:
```

```
procedure FIND_COMMON_ATTRIBUTE(input: request table;
                                output: attribute name);
    /*****************************************************
     * This procedure is used to find the name of the    *
     * jcin   attribute.                                 *
     * The join attribute  is the first attribute of the *
     * target list, sc we can just go to the entry       *
     * where the target list begins and extract the first*
     * attribute name and then return it to the calling  *
     * procedure.                                        *
     *****************************************************/
end procedure FIND_CCMMON_ATTRIBUTE;



procedure ALLOCATE_BUFFER(input: request_id;
                          output:hash_buffer);
    /*****************************************************/
    /* This procedure is used to allocate a buffer for   */
    /* storing the records and their hashed bucket number,*/
    /* set the length of the buffer to 0, and then       */
    /* return the buffer to the calling procedure.       */
    /*                                                   */
    /* The data structures and the variables used in     */
    /* this procedure are:                               */
    /*   1. hash_buffer:                                 */
    /*      A variable of the data type hashing_buffer,  */
    /*      which is defined in hashing_module.def       */
    /*      (see Appendix G).                            */
    /*   2. HB_ptr:                                      */
    /*      A pointer to the hash_buffer.                */
    /*   3. HB_id:                                       */
```

```
    /*      A field name of the hash_buffer that          */
    /*      contains the request id of the records        */
    /*      which belcng to this buffer.                  */
    /*****************************************************/
    HE_ptr = allocate the hash buffer;
    HE_ptr->HB_id = request_id;
    HE_ptr->length = 0;
end procedure ALLOCATE_BUFFER;



procedure STRING_HASH(input: result buffer, h_buffer);
    /*****************************************************
     * This procedure is called when the value type      *
     * of the common attribute is a character string.    *
     * It performs the  following tasks:                 *
     *  1. Extract records from the input result buffer  *
     *     one at a time.                                *
     *  2. Extract the value of the join  attribute      *
     *     from the extracted record and then check the  *
     *     lookup table to get the bucket number for     *
     *     the record.                                   *
     *  3. Store the bucket number and the record into   *
     *     a reserved hash buffer, h_buffer.             *
     *  4. If the hash buffer is full, then  send  the   *
     *     hash buffer to Bucket-block tracking          *
     *     procedure.                                    *
     *                                                   *
     * Data structures and variables used in this        *
     * procedure are:                                    *
     *  1. attribute_value: A character-string           *
     *                      representation of the common *
     *                      attribute value.             *
     *  2. record:  A character-string representation    *
```

```
*              of the extracted record.                *
*  3. bucket_number: The bucket number where the       *
*                    record characterized by the       *
*                    common attribute value is         *
*                    hashed into.                      *
*  4. bucket:  A character-string representation       *
*              of the bucket_number.                   *
*  5. EOV: The end-of-value marker.                    *
*  6. EON: The end-of-name marker.                     *
*  7. EOB: The end-of-buffer marker.                   *
*  8. LAST_RECORD: A boolean variable to indicate      *
*                  that this record is the last        *
*                  record for the request.             *
*  9. i: The index for the length of the result        *
*        buffer.                                       *
*     j: A general purpose index.                      *
*  10. lookup: The lookup table, which is an array     *
*              with 2048 character-string elements.    *
*       ---------------------                          *
*      |   0   |    abal     |                         *
*      |-------|-------------|                         *
*      |   1   |    abc      |                         *
*      |-------|-------------|                         *
*      |       :             |                         *
*      |       .             |                         *
*      |-------|-------------|                         *
*      |  2047 |   zyth      |                         *
*       ---------------------                          *
*                                                      *
*  11. h_buffer: A  variable of type hash_buffer       *
*                which is defined in                   *
*                hashing_module.def (see Appendix G)   *
*                and is used to store records and      *
*                their hashed values.                  *
*******************************************************/
```

```
/* Get the lookup table.                                    */
i = 1;
j = 0;
LAST_RECORD = false;
/* Get records from the result buffer one at a time. */
while result_buffer[i] <> EOB do
  /* Bypass the name of the common attribute.          */
  while result_buffer[i] <> EON do
    i = i+1;
  end while; /* Now, result_buffer[i] = EON.          */
  i = i+1;
  /* Get the value of the join attribute.             */
  While result_buffer[i] <> EOV do
    attribute_value[j] = result_buffer[i];
    i = i+1;
    j = j+1;
  end while; /* Now, result_buffer[i] = EOV.          */
  /* Compare the common attribute value with          */
  /* the contents of the lookup table to get the      */
  /* bucket-number.                                   */
  bucket_numbers = BI_SEARCH(lookup, attribute_number);
  perform NUMBER_TO_STRING(bucket_number, bucket);
  /* Add a EOV marker to the end of
     the attribute value. */
  attribute_value[j] = EOV
  /* Extract records from the buffer. */
  i = i+1;
  j = 0;
  repeat
    record[j] = result_buffer[i];
    i = i+1;
    j = j+1;
  until result_buffer[i-1] = EORecord;
  /* Now, record[j] = EORecord. */
  if result_buffer[i] = EORequest
```

```
        then
          LAST_RECORD = true;
          i = i+1;
      end if;
      /* Store the hashed information into the
         hash buffer, h_buffer.  */
      perform PUT_HASH_BUFFER(h_buffer, bucket,
                              attribute_value, record,
                              IAST_RECORD);
    end while;
end procedure STRING_HASH;



procedure PUT_HASH_BUFFER(input: h_buffer,
                                 bucket
                                 attribute_value, record,
                                 LAST_RECORD;
                          output: h_buffer);
    /*****************************************************
     *  This procedure is used to store the hashed       *
     *  record information into the hash_buffer.         *
     *                                                   *
     *  Data structures and variables used in this       *
     *  procedure are:                                   *
     *    1. X,Y,Z,i,j,K: General purpose indexes.       *
     *    2. MAX: The predefined maximum length of the   *
     *            hash buffer.                           *
     *    3. bucket: A character-string representation   *
     *               of bucket_number.                   *
     *    4. record: The input record which is in the    *
     *               form of character string.           *
     *    5. LAST_RECCRD: A boolean variable which is    *
```

```
 *                    used to indicate the end of    *
 *                    this request.                  *
 *     6. h_buffer: A buffer which is used to store  *
 *                  records and their hashed values. *
 ***************************************************/

/* Check to see if the buffer has enough space for  */
/* the new record. */
X = String_len(bucket_number);
Y = String_len(attribute_value);
Z = String_len(record);
K = the_current_length_of_the_hash_buffer;
if (K + X + Y + Z) > MAX
   then
     /* The buffer is  full, so it is send to the  */
     /* bucket-block tracking procedure.           */
     perform BUCKET_BLOCK(h_buffer);
     /* Reset the length of the buffer to 0. */
     K = 0;
   else
     /* The buffer has enough space, so store the  */
     /* input record into the buffer.*/
     for i = 1 to X do
        K = K + 1;
        hash_result[K] = bucket[i];
     end for;
     for i = 1 to Y do
        K = K + 1;
        hash_result[K] = attribute_value[i];
     end for;
     for i = 1 to Z do
        K = K + 1;
        hash_result[K] = record[i];
     end for;
     /* If this is the last record of this request,  */
```

```
        /* then send the hash_buffer to the                */
        /* bucket_block tracking procedure.                */
        if LAST_RECORD
           then
             hash_result[K+1] = EORequest;
             hash_result[K+2] = EOB;
             perform BUCKET_BIOCK(h_buffer);
             perform FREE_BUFFER_SPACE(h_buffer);
        end if;
      end if;
    end;
end procedure PUT_HASH_BUFFER;



procedure SMALL_INTEGER_HASH(input: result_buffer,
                                    MIN,
                                    h_buffer;
                             output:h_buffer);
    /*****************************************************
     * This procedure is used when the type of the       *
     * common attribute value is integer and when the    *
     * difference of the maximum and minimum value of    *
     * the common attribute value is less than the       *
     * number of the buckets of the hashing table.       *
     * It performs the following tasks:                  *
     *  1. Extract records from the input result buffer  *
     *     one at a time.                                *
     *  2. Extract the value of the common attribute from*
     *     the  extracted record and then calculate      *
     *     the bucket number.                            *
     *  3. Store the bucket number and the record into   *
```

```
*     a reserved hash-buffer.                                    *
* Data structures and variables used in this                     *
* procedure are:                                                 *
*  1. attribute_value: A character-string                        *
*                      representation of the common              *
*                      attribute value.                          *
*  2. record:  A character-string representation                 *
*              of the extracted record.                          *
*  3. bucket_number: The bucket number where the                 *
*                    record characterized by the                 *
*                    common attribute value is                   *
*                    hashed into.                                *
*  4. bucket:  A character-string representation                 *
*              of the bucket_number.                             *
*  5. EOV: The end-of-value marker.                              *
*  6. EON: The end-of-name marker.                               *
*  7. EOB: The end-of-buffer marker.                             *
*  8. LAST_RECORD: A boolean variable to indicate                *
*                  that this record is the last                  *
*                  record for the request.                       *
*  9. i: The index for the length of the result                  *
*        buffer.                                                 *
*     j: A general purpose index.                                *
*     k: The index for the length of the attribute_              *
*        value.                                                  *
*  10. temp: An integer representation of the input              *
*            attribute_value.                                    *
*  11. h_buffer: An variable of type hash_buffer                 *
*                which is defined in                             *
*                hashing_module.def (see Appendix G)             *
*                and is used to store records and                *
*                their hashed values.                            *
*************************************************************/

/* Initialize the indexes. */
```

```
i = 1;
k = 1;
j = 0;
LAST_RECORD = false;
/* Get the records from the result buffer
   one at a time.  */
while result_buffer[i] <> EOB do
  /* Bypass the name of the common attribute. */
  while result_buffer[i] <> EON do
    i = i+1;
  end while; /* Now, result_buffer[i] is EON. */
  i = i+1;
  /* Get the value of the common attribute. */
  while result_buffer[i] <> EOV do
    attribute_value[k] = result_buffer[i];
    i = i+1;
    j = j+1;
  end while; /* Now, result_buffer[i] is EOV. */
  /* Compute the bucket number.  */
  perform STRING_TO_NUMBER(attribute_value, Temp);
  bucket_number = Temp - MIN;
  perform NUMBER_TO_STRING(bucket_number, bucket);
  /* Add a EOV marker to the end of attribute value. */
  attribute_value[j] = EOV
  /* Get the attribute-value pairs of the actual  */
  /* target list of the record. */
  i = i+1;
  j = 0;
  repeat
    record[j] = result_buffer[i];
    i = i+1;
    j = j+1;
  until result_buffer[i-1] = EORecord;
  /* Now, record[j] is EORecord.  */
  if result_buffer[i] = EORequest
```

```
          then
            LAST_RECORD = true;
            i = i+1;
        end if;
        /* Store the hashed information into the h_buffer. */
        perform PUT_HASH_BUFFER(h_buffer, bucket,
                                attribute_number, record,
                                LAST_RECORD);
    end while;
end procedure SMALL_INTEGER_HASH;


procedure LARGE_INTEGER_HASH(input: result_buffer,
                                    MIN, range,
                                    h_buffer;
                             output:hash_buffer);
    /*****************************************************
     * This procedure is used when the type of the       *
     * common attribute value is integer and when the    *
     * difference of the maximum and minimum value of    *
     * the common attribute value is greater than the    *
     * number of the buckets of the hashing table.       *
     * It performs the following tasks:                  *
     *  1. Extract records from the input result buffer  *
     *     one at a time.                                *
     *  2. Extract the value of the common attribute from*
     *     the  extracted record and then calculate      *
     *     the bucket number.                            *
     *  3. Store the bucket number and the record into   *
     *     a reserved hash-buffer.                       *
     * Data structures and variables used in this        *
     * procedure are:                                    *
     *  1. attribute_value: A character-string           *
     *                      representation of the common *
```

```
*                        attribute value.                    *
*   2. record:  A character-string representation            *
*               of the extracted record.                     *
*   3. bucket_number: The bucket number where the            *
*                     record characterized by the            *
*                     common attribute value is              *
*                     hashed into.                           *
*   4. bucket:  A character-string representation            *
*               of the bucket_number.                        *
*   5. EOV: The end-of-value marker.                         *
*   6. EON: The end-of-name marker.                          *
*   7. EOB: The end-of-buffer marker.                        *
*   8. LAST_RECORD: A boolean variable to indicate           *
*                   that this record is the last             *
*                   record for the request.                  *
*   9. i: The index for the length of the result             *
*         buffer.                                            *
*      j: A general purpose index.                           *
*      k: The index for the length of the attribute_         *
*         value.                                             *
*   10. temp: An integer representation of the input         *
*             attribute_value.                               *
*   11. h_buffer: An variable of type hash_buffer            *
*                 which is defined in                        *
*                 hashing_module.def (see Appendix G)        *
*                 and is used to store records and           *
*                 their hashed values.                       *
*************************************************************/

/* Initialize the indexes.  */
i = 1;
k = 1;
j = 0;
LAST_RECORD = false;
/* Get records from the result buffer one at a time. */
```

```
while result_buffer[i] <> EOB do
  /* Bypass the name of the common attribute. */
  while result_buffer[i] <> EON do
    i = i+1;
  end while; /* Now, result_buffer[i] is EON. */
  i = i+1;
  /* Get the value of the join attribute. */
  while result_buffer[i] <> EOV do
    attribute_value[k] = result_buffer[i];
    i = i+1;
    j = j+1;
  end while; /* Now, result_buffer[i] is EOV. */
  /* Compute the bucket number.  */
  perform STRING_TO_NUMBER(attribute_value, Temp);
  bucket_value = TRUNC[(Temp - MIN)/range];
  perform NUMBER_TO_STRING(bucket_value, bucket);
  /* Add a EOV marker to the end of attribute_value. */
  attribute_number[j] = EOV
  /* Get the attribute-value pairs of the actual     */
  /* target list of the record.  */
  i = i+1;
  j = 0;
  repeat
    record[j] = result_buffer[i];
    i = i+1;
    j = j+1;
  until result_buffer[i-1] = EORecord;
  /* Now, record[j] is EORecord. */
  if result_buffer[i] = EORequest
     then
       LAST_RECORD = true;
       i = i+1;
  end if;
  /* Store the hashed information into the h_buffer. */
  perform PUT_HASH_BUFFER(h_buffer, bucket,
```

```
                                attribute_number, record,
                                LAST_RECORD);
    end while;
end procedure LARGE_INTEGER_HASH;
```

APPENDIX E

THE BUCKET-BLOCK-TRACKING PROCEDURE PROGRAM SPECIFICATIONS

```
procedure BUCKET_BLOCK(input: H_buffer);
    /*****************************************************
     * This procedure receives a hash buffer, H_buffer,  *
     * from the ret_com subfunction and performs the     *
     * following task.                                   *
     *   1. Establish and maintain  a global table to    *
     *      store the addresses of the hashing tables    *
     *      of all the requests.                         *
     *   2. Extract the hashed record  information from  *
     *      the input hash_buffer.                       *
     *   3. Check the global table to see if the input   *
     *      records belong to a new request. If they do, *
     *      then allocate a new hashing table.           *
     *      Otherwise, get the logical address of the    *
     *      hashing table from the global table and      *
     *      assign a pointer to the hashing table.       *
     *   4. Group records into the buckets according to  *
     *      their bucket numbers and store them into     *
     *      blocks.                                      *
     *   5. Broadcast the bucket information of the local *
     *      target records to the other backends.        *
     *   6. Store the hashing table back to the secondary *
     *      storage.                                     *
     *                                                   *
     *  Data structures and variables used in this       *
     *  procedure are:                                   *
     *                                                   *
     *  1. FIRST_RET_COM :                               *
     *        A boolean variable which is set to         *
     *        true when the first retrieve common        *
```

```
*       request enters the system.                          *
*                                                           *
*  2. GT_ptr:                                               *
*       A pointer to a global table.                        *
*  3. G_table:                                              *
*       A variable of type global table (see                *
*       Appendix G).                                        *
*                                                           *
*  4. HT_ptr:                                               *
*       A pointer to a hashing table.                       *
*  5. HT:                                                   *
*       A variable of type Hash_table (see                  *
*       Appendix G).                                        *
*                                                           *
*  6. HB_ptr:                                               *
*       A pointer to a hash buffer.                         *
*  7. H_buffer:                                             *
*       A variable of type hash_buffer (see                 *
*       Appendix G).                                        *
*                                                           *
*  8. NEW_REQUEST:                                          *
*       A boolean variable which is set to                  *
*       true if the request id cannot be found              *
*       in the global table.                                *
*  9. logical_addr:                                         *
*       A variable of type addr_definition,                 *
*       which is defined in the commdata.def file.          *
*  10. bucket_number:                                       *
*       The bucket number where the record                  *
*       characterized by the attribute value is             *
*       hashed into.                                        *
*  11. bucket:                                              *
*       A character-string representation of                *
*       the bucket_number.                                  *
*  12. req_id:                                              *
```

```
    *        A  record which contains the traffic id and *
    *        request number of a request.                *
    *  13. i, j:                                         *
    *        General purpose indexes.                    *
    ******************************************************/

if FIRST_RET_COM
   then
     perform INITIALIZE_GLOBAL_TABLE(GT_ptr);
     FIRST_RET_COM = false;
end if;

  /* Get the request id from the pointer of which       */
  /* points the input hash buffer.                      */
  request_id = H_buffer.Request_id;

  /* Check the global table to see if this request is   */
  /* a new request.                                     */
  perform CHECK_GLOBAL_TABLE(GT_ptr, req_id,
                             logical_addr, NEW_REQUEST);
  if NEW_REQUEST
     then
       perform ALLOCATE_HASH_TABLE(logical_addr);
       perform INSERT_GLOBAL_TABLE(GT_ptr, req_id,
                                   logical_addr);
  end if;
  perform GET_HASHING_TABLE(request_id,
                            logical_addr, HT);

  /* Now, the hashing table is ready to store records. */
  /* Extract the record information from the           */
  /* hash buffer one record at a time.                 */
  /* Because the last two character of the hash buffer */
  /* are the EORequest marker which indicates whether  */
  /* this is the last hash buffer for this request     */
  /* and the EOBuffer marker which indicates the       */
  /* end of this hash buffer, the actual length of the */
```

```
/* hash buffer is length-2.                                    */
j = 1;
while j ≤ (H_buffer.length-2) do
  /* Get the bucket number. */
  i = 0;
  repeat
    bucket[i] = H_buffer.Hashed_result[j];
    i = i +1;
    j = j + 1;
  until H_buffer.Hashed_result[j] = EOV;

  /* Convert the bucket number from a character to  */
  /* an integer. */
  bucket_number = STRING_TO_INTEGER(bucket);
  /* Get the common attribute value and the record   */
  /* itself. */
  j = j + 1;
  i = 0;
  repeat
    common_and_record[i] = Hash_buffer.HB_buffer[j];
    i = i + 1;
    j = j + 1;
  until common_and_record [i - 1] = EORecord;

  /* Store the record and its common attribute value */
  /* into the hashing table.    */
  perform STORE_RECORD_IN_HASH_TABLE(HT, bucket_number,
                                     common_and_record,
                                     NEW_RECORD);

  NEW_RECORD = false;
end while;

/* Check if this is target request                        */
if MOD(req_id.request_no, 2) = 0
   then
     /*  This is a target request.                        */
```

```
        perform BROADCAST_TARGET_INFO(HT);
    end if;
    perform STORE_BACK(HT, logical_addr)
end procedure BUCKET_BLOCK;



procedure INITIALIZE_GLOBAL_TABLE(output: GT_ptr);
    /*****************************************************
     *  This procedure is used when the first retrieve-  *
     *  common request is executed in the BUCKET_BLOCK   *
     *  procedure.                                       *
     *  This procedure creates a global table and        *
     *  returns the pointer (GT_ptr)  to the table to    *
     *  the calling procedure.                           *
     *****************************************************/
end procedure INITIALIZE_GLOBAL_TABLE;



procedure ALLOCATE_HASH_TABLE(output: logical_addr);
    /*****************************************************
     *  This procedure is used to allocate a hashing     *
     *  table for a new retrieve-common request from     *
     *  a predefined secondary storage area and return   *
     *  the logical disk address to the calling          *
     *  procedure.                                       *
     *  The bucket entries are also initialized.         *
     *****************************************************/
end procedure ALLOCATE_HASH_TABLE;
```

```
procedure CHECK_GLOBAL_TABLE(input: GT_ptr, request_id;
                    output: logical_addr, NEW_REQUEST);
    /*****************************************************
     *  This procedure is used to check whether a request *
     *  is a new request  by checking its request id      *
     *  against the global table. If the request id is    *
     *  found in the global table, then set the value of  *
     *  NEW_REQUEST to false and return the logical disk  *
     *  address of  the  hashing  table  to the calling   *
     *  procedure.  Otherwise, return the NEW_REQUEST     *
     *  back to the calling procedure.                    *
     *****************************************************/
end procedure CHECK_GLOBAL_TABLE;


procedure INSERT_GLOBAL_TABLE(input: GT_ptr, Req_id,
                                     logical_addr;
                              output: GT_ptr);
    /*****************************************************
     *  This procedure is used to insert a  new hashing  *
     *  table into the global table.                      *
     *                                                    *
     *  Data structures and variables used in this        *
     *  procedure are:                                    *
     *    1. GT_ptr:                                      *
     *       A pointer to the global table.               *
     *    2. Req_id:                                      *
     *       The request id of the records of the new     *
     *       hashing table.                               *
     *    3. logical_addr:                                *
     *       The logical disk address of the new hashing  *
     *       table.                                       *
```

```
     *                                                    *
     *  An  inverted list implementation to maintain the  *
     *  table is recommanded.                             *
     ******************************************************/
end procedure INSERT_GLOBAL_TABLE;



procedure STORE_RECORD_IN_HASH_TABLE
                        (input: HT, bucket_number,
                                info, NEW_RECORD);
    /*****************************************************
     *  This procedure is used to store the common        *
     *  attribute value of a record and the record itself *
     *  into a hashing table.                             *
     *  Recall that the records are stored in blocks.     *
     *                                                    *
     *  Data structures and the variables used in this    *
     *  procedure are:                                    *
     *    1. HT:                                          *
     *       A variable of type hash_table which is       *
     *       defined in  hashing_module.def (see Appendix *
     *       G).                                          *
     *    2. bucket_number:                               *
     *       The bucket number where the record           *
     *       characterized by the common attribute value  *
     *       is hashed into.                              *
     *    3. info:                                        *
     *       A character string which contains the        *
     *       common attribute value of a record and the   *
     *       record itself.                               *
     *    4. NEW_RECORD:                                  *
     *       A boolean variable to indicate whether the   *
```

```
*       input info is a new record of this request   *
*       id.                                          *
*    5. old_bucket_number:                           *
*       The bucket_number of the previous input      *
*       record.                                      *
*    6. bkt:                                         *
*       A variable of type BUCKET_ENTRY which is     *
*       defined in hashing_module.def (see Appendix  *
*       G).                                          *
*    7. blk_ptr:                                     *
*       A pointer to a record block of type          *
*       REC_BLOCK which is defined in                *
*       hashing_module.def (see Appendix G).         *
*    8. blk, blk_2:                                  *
*       Variables of type REC_BLOCK which is defined *
*       hashing_module.def (see appendix G).         *
*    9. I:                                           *
*       An integer variable.                         *
*    10. MAX_BLOCK_SIZE:                             *
*        An integer that represent the maximum       *
*        length of the block content.                *
**************************************************/

if NEW_RECORD
   then
     /* This record is the first input record of this */
     /* request.                                      */
     perform GET_THE_BUCKET(HT, bucket_number, bkt);
     perform ALLOCATE_REC_BLOCK(blk);
     perform MODIFY_ENTRY_&_HEADER(bkt, blk);
   else
     /* Compare the input bucket_number with the
        previous one.  */
     if bucket_number <> old_bucket_number
        then
```

```
            perform STORE_BACK(blk);
            /* Get the desired bucket entry  for this
               input record. */
            bkt = HT.bkt_entries[ bucket_number ];
            /* Check if the bucket is empty. */
            if bkt.status = empty
               then
                 perform ALLOCATE_REC_BLOCK(blk, addr);
                 perform MODIFY_ENTRY_&_HEADER(bkt,
                                               blk,addr);
               else
                 /* Get the record block by the address */
                 /* in the bucket entry.*/
                 perform GET_REC_BLOCK(bkt.block_address,
                                                  blk);
            end if;
         end if;
         /* Check if the block has enough space to  */
         /* store this record. */
         I = STRING_IENGTH(info);
         if (blk.header.length + I) > MAX_BLK_SIZE
            then
              /* This block does not have enough space */
              /* for this record. */
              perform ALLOCATE_RECORD_BLOCK(blk_2,
                                            addr_2);
              perform MODIFY_ENTRY_&_HEADER(bkt,
                                            blk_2,
                                            addr_2);
                      /* This routine will also modify */
                      /* the header of blk_2. */
               perform STORE_BACK(blk);
               blk = blk_2;
          end if;
end if;
```

```
    perform STORE_INFO_IN_BLOCK(info, blk);
end procedure STORE_RECORD_IN_HASH_TABLE;



procedure STORE_BACK(input: A_structure);
    /*****************************************************
     * This procedure is used to store a hashing table, *
     * or a record block back to the secondary storage. *
     *                                                   *
     * A_structure  is  a variable which may be either  *
     *               a hashing table or a block.         *
     *****************************************************/
end procedure STORE_BACK;


procedure GET_REC_BLOCK(input: logical_addr;
                        output: blk);
    /*****************************************************
     * This procedure is used to bring a block of memory *
     * from a predefined secondary storage area into the *
     * primary memory by its logical address.            *
     * Data structures and variables used in this        *
     * procedure are:                                    *
     *   1. logical_addr                                 *
     *      The logical address of a block.              *
     *      A variable of addr_definition which is       *
     *      defined in the commdata.def file.            *
     *   2. blk                                          *
     *      A variable of type REC_BLOCK which is defined*
     *      in  the hashing_module.def (see Appendix G). *
     *****************************************************/
end procedure GET_REC_BLOCK;
```

```
procedure STORE_INFO_IN_BLOCK(input: info, blk);
    /*****************************************************
     *  This procedure is used to store the common       *
     *  attribute value of a record and the record       *
     *  itself into a block.                             *
     *  It is called only when the block has enough      *
     *  space for that information, i.e., info.          *
     *  Data structures and variables used in this       *
     *  procedure are:                                   *
     *    1. info:                                       *
     *       A character string which contains the       *
     *       common attribute value of a record and      *
     *       the  reccrd itself.                         *
     *    2. blk:                                        *
     *       A variable of type REC_BLOCK which is       *
     *       defined in hashing_module.def (see          *
     *       Appendix G).                                *
     *    3. i,j:                                        *
     *       General purpose indexes.                    *
     *****************************************************/

    i = 0;
    j = blk.header.length+1;
    repeat
      blk.contents[j] = info[i];
      i = i+1;
      j = j+1;
    until i = STRING_IENGTH(info);
end procedure STORE_INFO_IN_BLCCK;
```

```
procedure MODIFY_ENTRY_&_HEADER(input: bkt, blk,
                                       blk_addr;
                                output: bkt, blk);
    /*****************************************************
     *  This procedure is used to modify the bucket      *
     *  entry of the input  bkt  and the header part     *
     *  of the input  blk.  It will then return these    *
     *  modified  bkt  and  blk  back to the calling     *
     *  procedure.                                       *
     *                                                   *
     *  Data structures and variables used in this       *
     *  procedure:                                       *
     *    1. bkt:                                        *
     *       A variable of type Bucket_entry             *
     *       which is defined in hashing_module.def      *
     *       (see Appendix G).                           *
     *    2. blk:                                        *
     *       A variable of  type REC_BLOCK which         *
     *       is defined in hashing_module.def            *
     *       (see Appendix G).                           *
     *    3. blk_addr                                    *
     *       A variable of type  addr_definition         *
     *       which is the logical address of a block     *
     *       and is defined in the commdata.def file.    *
     *****************************************************/

    blk.header.next_blk_addr = bkt.block_address;
    bkt.block_address = blk_addr;
end procedure MODIFY_ENTRY_&_HEADER;
```

```
procedure BROADCAST_TARGET_INFO(input: HT);
    /*****************************************************
    *  This procedure is used to broadcast the records  *
    *  of the target hashing table to the other         *
    *  backends.                                        *
    *  This is the same procedure that  is used to      *
    *  broadcast the descriptor ids among backends.     *
    *  Data structures and variables used in this       *
    *  procedure are:                                   *
    *  1. HT:                                           *
    *     A variable of type hashing_table              *
    *     which is defined in hashing_module.def        *
    *     (see Appendix G).                             *
    *  2. i:                                            *
    *     A general purpose index.                      *
    *  3. MAX_BKT_#:                                    *
    *     An integer which is used to represent the     *
    *     maximum number of the bucket entries in a     *
    *     hashing table.                                *
    *  4. bkt:                                          *
    *     A variable of  type  Bucket_entry  which      *
    *     is defined in hashing_module.def (see         *
    *     Appendix G).                                  *
    *  5. msg:                                          *
    *     A  character string  which is used to store   *
    *     the message that  is to be broadcasted to all *
    *     of the  backends.                             *
    *****************************************************/

   for i = 1 to MAX_BKT_# do
       bkt = HT.bkt_entries[i];
       if bkt.status <> empty
          then
            /* Put the bucket number into  the message.*/
            perform GET_REC_BLOCK(bkt.block_address,blk);
```

```
            repeat
              /* Extract the contents of the  */
              /* blk.content and copy them into msg.*/
                 if the msg is full
                    then
                      send msg to all of the backends;
                      reset the length of msg to 0;
                 end if;
              if blk.next_blk_address = blk.own_address
                 then
                   /* This block is the last block for
                      this bucket. */
                   last = true;
            until last;
        end if;
     end for;
     send the msg to all of the other backends;
end procedure BROADCAST_TARGET_INFO;
```

APPENDIX F

THE MERGING PROCEDURE PROGRAM SPECIFICATIONS

```
procedure MERGE(input: source_request_id,
                       logical_address_of_source_table,
                       logical_address_of_target_table);
   /*****************************************************
    *                                                   *
    * This procedure is used to perform the merging     *
    *                                                   *
    * operation over the source records and the target  *
    *                                                   *
    * records.                                          *
    *                                                   *
    * Notice that the input addresses are the logical   *
    *                                                   *
    * disk addresses of the   two hashing tables.       *
    *                                                   *
    * Data structures: and variables used in this       *
    *                                                   *
    * procedure are:                                    *
    *                                                   *
    *   1. logical_address_of_source_table,             *
    *                                                   *
    *      logical_address_of_target_table:             *
    *                                                   *
    *      The  logical  disk  addresses of the source  *
    *                                                   *
    *      and the target hashing tables, both of the type*
    *                                                   *
    *      address definition which is defined in the   *
    *                                                   *
    *      commdata.def file.                           *
    *                                                   *
    *   2. source_table, target_table:                  *
    *                                                   *
    *      Variables of hashing_table data type (see    *
    *                                                   *
    *      Appendix G) that represents the source-hashing *
    *                                                   *
    *      table and the target-hashing table.          *
    *                                                   *
    *   3. i: A general purpose index.                  *
    *                                                   *
    *   4. max_bucket_number:                           *
    *                                                   *
    *      The largest bucket number of a hashing table. *
    *                                                   *
    *****************************************************/
    /* Retrieve the   two hashing tables by the input    */
    /* logical addresses.                                */
    /* Ncte: Due to the limited memory space, we may     */
    /*       not be able to bring in the entire table.   */
    perform GET_HASH_TABLE(logical_address_of_source_table,
```

```
                                source_table);
    perform GET_HASH_TABLE(logical_address_of_target_table,
                                target_table);
    /* Reserve a result buffer. */
    perform GET_BUFFER(result_buffer,source_request_id);
       /* This routine will allocate an instance of a
          result buffer and put the request id into the
          the header of the buffer and initialize the
          length of the buffer to 0.
          This routine has already been coded in
          the retp.c file. */
    i = 0;
    while i ≤ max_bucket_number do
      if [(source_table.bucket_entry[i].status <> empty)
          and
          (target_table.bucket_entry[i].status <> empty)]
          then
            /* There is a collision. */
            /* Retrieve the records from both blocks and
               perform the merging operation. */
            X = source_table.bucket_entry[i].logical_address;
            Y = target_table.bucket_entry[i].logical_address;
            perform merging_operation(X,Y,result_buffer);
              /* This routine will perform the merging
                 operation and send the merged results
                 to the controller. */
      end if;
      i = i+1;
    end while;
    /* Signal  PP upon the completion of the source and */
    /* target request. */
end procedure MERGE;
```

```
procedure MERGING_OPERATION
            (input: logicl_address_source_block,
                    logicl_address_target_block,
                    result_buffer;
             output: result_buffer);
    /*****************************************************
     *  This procedure is used to perform the following   *
     *  tasks:                                            *
     *    1. Extract the records from both of the source  *
     *       block and the target block.                  *
     *    2. Compare the common attribute values          *
     *       of the source and target records.            *
     *       If they are equal, then perform the merging  *
     *       operation.                                   *
     *    3. Put the merged results into a result buffer. *
     *       If the buffer is full, then send the buffer  *
     *       to the controller and reinitialize the       *
     *       buffer length to 0 so that the buffer can    *
     *       be reused.                                   *
     *       Otherwise, return the logical address of the *
     *       the result buffer to the calling procedure.  *
     *                                                    *
     *  Data structures and variables used in this        *
     *  procedure are:                                    *
     *    1. source_block, target_block:                  *
     *       Variables of the data type  BKT_BLK  which   *
     *       are used to represent the blocks of the      *
     *       source hashing table or the target hashing   *
     *       table.                                       *
     *       BKT_BLK  is defined in hashing_module.def    *
     *       (see Appendix G).                            *
     *    2. source_done, target_done:                    *
     *       Boolean variables which are used to indicate *
     *       the completion of processing either source   *
```

```
 *       records or target records.                    *
 *    3. i,j: General purpose indexes.                 *
 ****************************************************/
/* Continue retrieving  the source blocks by the    */
/* logical address, until there are no more blocks. */
repeat
  source_block =
            GET_BLOCK(logical_address_source_block);
  /* Continue retrieving the target blocks by the  */
  /* logical address until there are no more blocks.*/
  repeat
    target_block =
            GET_BLOCK(logical_address_target_block);
    i = 0;
    while source_block.body[i] <> EOB do
      /* Retrieve one common attribute_value and one */
      /* record from source block.                   */
      source_value = GET_VALUE(source_block.body,i );
      source_record = GET_RECORD(source_block.body,i);
      J = 0;
      while target_block.body[j] <> EOB do
        /* Retrieve one common attribute_value and  */
        /* one record from the target block.       */
        target_value = GET_VALUE(target_block.body, j);
        target_record =
                  GET_RECORD(target_block.body,j);
        if source_value = target_value
           then
             /* Append target record at the end of  */
             /* source record and put the newly     */
             /* merged record into the result buffer.*/
             result = APPEND(source_record,
                             target_record);
             result_length = STRING_LENGTH(result);
             perfcrm RB$PUT_SEND(result_buffer,
```

```
                                        result,
                                        result_length);
                else
                  /* Go to the next target record. */
                  J = J+1;
              end if;
            end while; /* End the target-record loop. */
            i = I+1;
          end while; /* End the source-record loop.*/

          /* Are the target records done? */
          if target_block.header.next_block_address =
             target_block.header.this_block_address
             then
               target_done = true;
             else
               target_block.header.next_block_address =
                 target_block.header.this_block_address;
          end if;
        until target_done;

        /* Are the source records done? */
        if source_block.header.next_block_address =
           source_block.header.this_block_address
           then
             source_done = true;
           else
             source_block.header.next_block_address =
             source_block.header.this_block_address;
        end if;
      until source_done;
end procedure MERGING_OPERATION;
```

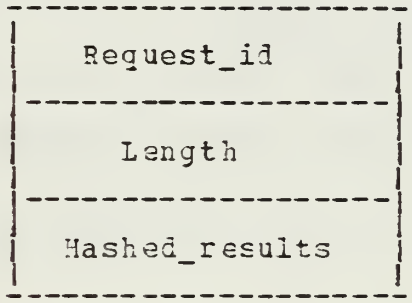
APPENDIX G

THE HASHING MODULE DATA STRUCTURE DEFINITIONS

In this appendix we present the definitions of the data structures used in the previous appendices. We refer to the definitions as hashing_module.def.

1. hash_buffer:

   This is the buffer which stores the hashed information of records.



--> The request id of the hashed records.

--> The current length of the Hashed_results.

--> An array of character string used for storing the hashed records.

The format of the hashed_results is:

{hashed_record_info}+ EOReq EOB

where

hashed_record_info :: = bucket_number EOV {Rec}+

Rec :: = {attribute_value_pair}+EORec

attribute_value_pair :: =

attribute_name EON attribute_value EOV

"+" means one or more occurence.

ECB : A special character which is used as a marker for the end-of-buffer.

ECV : A special character which is used as a marker for the end-of-value.

ECN : A special character which is used as a marker for the end-of-attribute_name.

EORec: A special character which is used as a marker

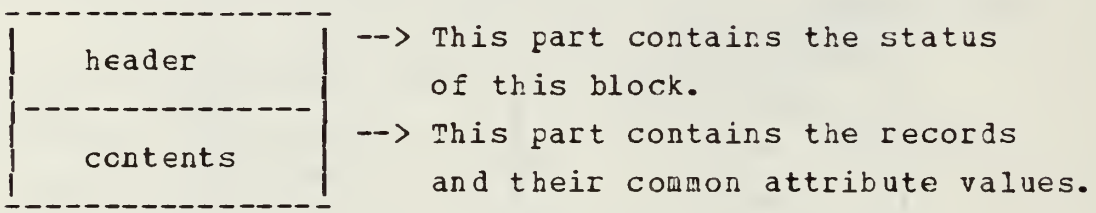
for the end-of-record.

ECReq: A character, either 1 or 0 , which is use to indicate the end of a request.
1: end of a request.
0: not end of request, more buffers are coming.

2. REC_BLOCK

Blocks used by buckets to store the records and their common attribute values.

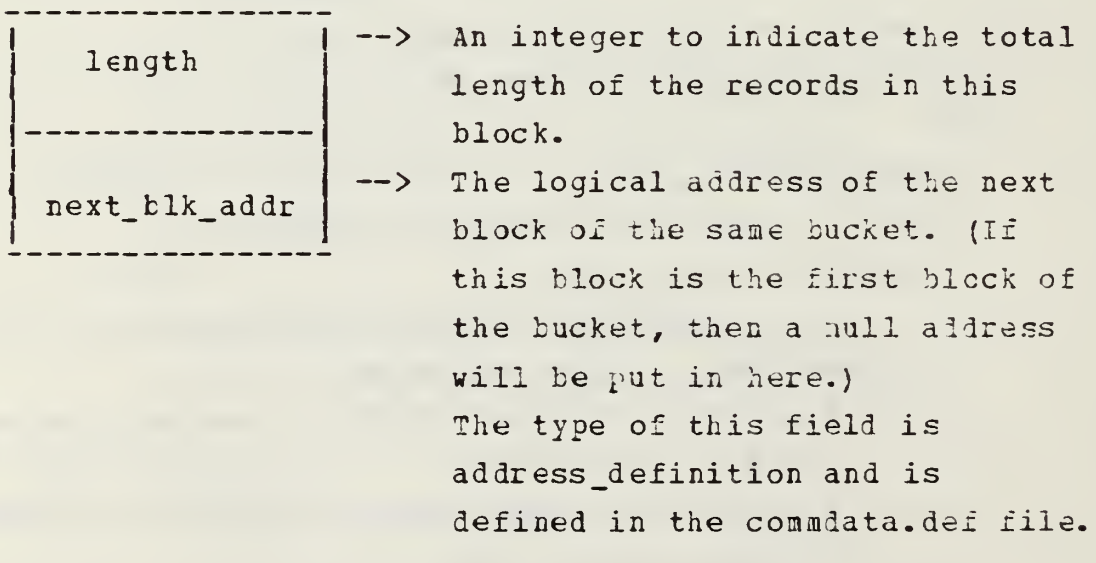
A REC_BLOCK is composed of a header two fields, and a contents.



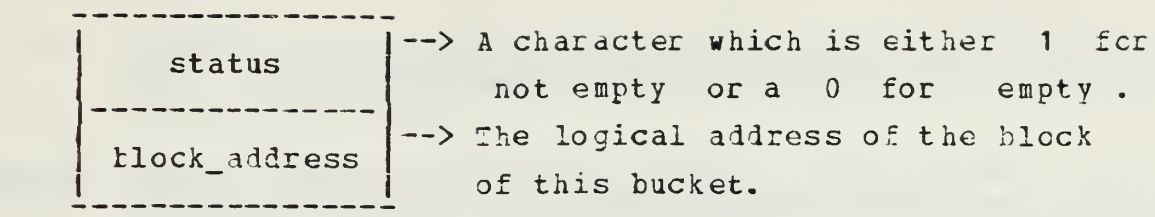
The format of the content of the REC_BLOCK is:

{Rec}+EOB

The header contains two parts:



3. Bucket_entry:



4. Hash_table: An array of 2048 bucket_entries.

LIST OF REFERENCES

1. Date, C. J., An Introduction To Database System Volume 1, Addison Wesley, 1982.

2. Lowenthal, E. I., "The Backend Computer, Part I and Part II," Auerbach (Data Management) Serious, 24-01-04 and 24-01-05, 1976

3. Maryanski, F. J., "Backend Database System", Computing Surveys, Vol. 12, No. 1, pp.3-25, March 1980.

4. Naval Postgraduate School Technical Report NPS52-83-006, Design and Analysis of a Multi-Backend Database System for Performance Improvement, Functionality Expansion and Capacity Growth by D. K. Hsiao and M. J. Menon, June 1983.

5. The Ohio State University Technical Report OSU-CISRC-TR-82-1, The Implementation of a Multi-Backend Database System (MBDS): Part I - Software Engineering Strategies and Efforts Towards a Prototype MBDS, by Kerr, D. S., and others, January 1982.

6. Hsiao, D. K. and Harary, F. A. "A Formal System for Information Retrieval from files," Communications of the ACM, Vol. 13, No. 2, pp.67-73, February 1970.

7. Naval Postgraduate School Technical Report NPS52-85-002, A Multi-Backend Database System for Performance Gains, Capacity Growth and Hardware Upgrade. by S. A. Demurjian and others, Feburary 1985.

8. Muldur, S., Design and Analysis of Ordering and Join Operations for a Multi-backend Database System Master Thesis, Naval Postgraduate School, Monterey, California, June 1984.

9. The Ohio State Jniversity Technical Report OSU-CISRC-TR-81-11, A Survey of Parallel Sorting Algorithms, by D. K. Hsiao and others, December 1981.

10. The Ohio State University Technical Report OSU-CISRC-TR-80-7, Parallel Record-Sorting Methods for Hardware Realization, by D. K. Hsiao and M. J. Menon, July 1980.

11. Naval Postgraduate School Technical Report NPS52-82-008, The Implementation of a Multi-Backend Database System (MBDS): Part II - The First Prototype MBDS and the Software Engineering Experience. By X.

He and the others, July 1982.

12. Johnson, S. C., "YACC: Yet Another Complier-Complier", UNIX, TIME-SHARING SYSTEM: UNIX PROGRAMMER'S MANUAL, Bell Telephone Laboratories, Incorporated, Murray Hill, N.J., 1982.

13. Naval Postgraduate School Technical Report NPS52-84-005, The Implementation of a Multi-Backend Database System (MBDS): Part IV - The Revised Concurrency Control and Directory Management Processes and the Revised Definitions of Inter-process and Inter-computer Messages by S. A. Demurjian and the others, February 1984.

INITIAL DISTRIBUTION LIST

No.		Copies
1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22304-6145	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, California 93943-5100	2
3.	Chairman, Code 052 Department of Computer Science Naval Postgraduate School Monterey, California 93943-5100	2
4.	Curriculum Officer, Code 037 Computer Technology Program Naval Postgraduate School Monterey, California 93943-5100	2
5.	Professor David K. Hsiao, Code 052 Department of Computer Science Naval Postgraduate School Monterey, California 93943-5100	2
6.	Steven A. Demurjian, Code 052 Department of Computer Science Naval Postgraduate School Monterey, California 93943-5100	2
7.	Hsiang-lung Tung 8, Lane 46, Ming-Chuan Road Chia-Yi City, Taiwan 600 Republic of China	2